ALGORITHMS FOR LARGE GRAPHS

A Thesis
Presented to
The Academic Faculty

by

Atish Das Sarma

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Algorithms, Combinatorics, and Optimization

Georgia Institute of Technology
August 2010

ALGORITHMS FOR LARGE GRAPHS

Approved by:

Professor Richard J. Lipton, Advisor
College of Computing
Georgia Institute of Technology

Professor Robin Thomas
School of Mathematics
Georgia Institute of Technology

Professor Gopal Pandurangan
Mathematical Sciences Division
Nanyang Technological University

Professor Santosh Vempala
College of Computing
Georgia Institute of Technology

Professor Dana Randall
College of Computing
Georgia Institute of Technology

Date Approved: 11 June 2010

ACKNOWLEDGEMENTS

I am grateful to my advisor Dick Lipton for his constant support, motivation and encouragement throughout my stay at Georgia Tech. His guidance has been invaluable in shaping much of my research. He has generously provided me with abundant freedom to work on a variety of research problems and has encouraged me to participate in all my collaborations. Working with him has been a genuine pleasure, and I have benefited immensely from having Dick as my advisor.

I would like to thank my collaborators Sreenivas Gollapudi, and Rina Panigrahy from Microsoft for our numerous interactions throughout my PhD. They have been wonderful collaborators, mentors, and friends. I owe a lot of my research to their constant support and guidance. Through the years, Sreenivas has introduced me to several researchers in the community. We have had many inspiring discussions and he has been a consistent source of good advice. Working with Rina on various collaborative projects has been fun, exciting, intimidating at times, and enriching throughout.

I am deeply thankful to my constant collaborator and fellow student Danupon Nanongkai. Research discussions with Danupon have been very fruitful and I have come to treasure our endless conversations while staring at and scribbling on a whiteboard. Danupon has also been an extremely close friend to me during these years.

I owe a lot of gratitude to my collaborators and mentors Gopal Pandurangan, Jim Xu, Prasad Tetali, and Ashwin Lall. Gopal's encouragement, drive, and enthusiasm, Jim's and Prasad's constant support, and Ashwin's help and guidance have made the last couple years a lot easier for me.

My close friends Gagan Goel, Subrahmanyam Kalyanasundaram, Himani Jain,

and Anirudh Ramachandran, with whom I have spent several memorable hours, deserve to be specially mentioned. I thank Gagan for being the best roommate, Subrahmanyam for being such a dependable lunch and gossip buddy, Himani for the wonderful times we shared, and Anirudh for all our conversations over beer. I also appreciate the cheerful support that many of my other friends have given me.

All the faculty and students in the theory group have helped in making my time as a graduate student productive. I am particularly indebted to Dick Lipton, Robin Thomas and the ACO, and Santosh Vempala and the ARC for their support, and research and travel funds. Many thanks to all my committee members, including Dana Randall for some initial discussions.

I thank my girlfriend Dyuti for always being with me, and for her support and encouragement. I draw great strength and inspiration from her presence.

Finally, I cannot thank enough my wonderful family without whom nothing I have achieved would be possible or even meaningful. My younger brother, Akash, has always been a great source of pride, joy, and encouragement. I am fortunate to have my twin brother Anish as a collaborator. His support and advice through the years have helped me find direction at several crucial stages. My parents have always stood by me from a distance, in many ways, and their affection has made my journey much more enjoyable. My father, an accomplished researcher himself, has taught me the value of believing in oneself. Through years of discussions about achieving and succeeding, he has helped in bringing out the best in me, My mother's ability to stay happy and to treasure life's important things have helped me maintain the right perspective at various times in my own life.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This thesis explores algorithms for massive graphs. With the advent of the internet, web users interact with large graphs on a daily basis. Some examples include the web graph, and social networks. Several other massive graphs such as query-document click graphs, or peer to peer networks are also indirectly used by hundreds of millions of users regularly. Several of these graphs are not only massive but also change at staggering rates. Analyzing such large graph data sets, while extremely useful in improving the interaction between users and the systems, calls for new and efficient techniques. In this thesis, we explore efficient graph algorithms under standard models of processing such large data sets.

Three specific well studied frameworks are considered in this thesis, all of which are now standard in the literature of dealing with massive graphs. The first framework considered is the streaming model. The basic premise of the the streaming model is that the input graph is too large to store in main memory and therefore is stored on disk. Any algorithm processing the input has sequential access; therefore, the algorithm proceeds by sequentially scanning the edges of the graph, while maintaing small memory/space requirement. Each sequential scan is referred to as a pass, and the space allowed is linear in the number of vertices. The goal is to minimize the number of passes required for the task. Several variants have been considered as to in what order the edges of the graph are accessed - in the most general setting, the order is assumed to be arbitrary.

The second framework considered is the distributed computing congest model popularly used for studying peer to peer networks. In this model, each node has local knowledge (of its neighbors), and any two neighboring nodes may communicate

along edges. Communication proceeds in rounds, with a bandwidth restriction of logarithmic bits per edge per round. The goal is for some decentralized computation to be performed through such local operations.

The third framework looked at in this thesis deals with answering queries online extremely efficiently. Given a massive graph, some pre-processing is allowed and a summary of the graph, or summary associated with all nodes (referred to as sketches), may be stored. Then when the query is issued, that pertains to certain nodes, the sketches of the relevant nodes can be retrieved which may then be aggregated to compute the required property. This computation also needs to be done very efficiently. There are bounds on the size of the pre-processed information (typically should not be much more than logarithmic bits per node), however, the time of the pre-processing itself can be relaxed.

The main graph problem considered in this thesis is performing random walks efficiently. Random walks are fundamental across all of computer science ranging from theory, mathematics, distributed computing and web algorithms. Several applications include search, data mining, ranking (such as PageRank), measuring similarity between web pages, maintaining connectivity in P2P networks etc. The work in this thesis has developed the fastest algorithms for performing random walks in the streaming model, and in distributed networks, breaking past the linear-time barrier presented by the sequential nature of random walks. This work improves upon techniques that have been used for decades in both theory as well as practice. This thesis also considers several other graph problems.

Specifically in the streaming model, we show how to perform several independent random walks of length ℓ in $O(\sqrt{\ell})$ rounds, improving upon the naive $O(\ell)$ round approach. We then extend this work to estimate probability distributions including steady state distributions of the random walk on undirected or directed graphs. Specifically, we show how to estimate PageRank vectors in $\tilde{O}(n)$ space and $\tilde{O}(\sqrt{M})$

passes, where M is the mixing time of the graph. PageRank was one of the pioneering ideas in the Google search engine. This work improves upon one of the standard implementations that requires $\tilde{O}(n)$ space and $\tilde{O}(M)$ passes. For another choice of parameters in our algorithm, we can estimate the PageRank vector in $\tilde{O}(nM^{-1/4})$ space and $\tilde{O}(M^{5/8})$ passes. Notice that this greatly improves upon both the space and pass requirements. This technique also first suggests how to estimate the mixing time itself and subsequently get approximations to related quantities such as the conductance of the graph.

In follow up work on the streaming model presented in this thesis, we consider the problem of graph partitioning. Several offline and few streaming/online algorithms have been suggested for partitioning an input graph into two parts, to approximate the conductance of the graph. While many of these techniques are impractical at the scale of the massive web graphs, some may be implementable. However, when dealing with several hundred million nodes and tens of billions of edges, visualizing one global cut becomes very difficult. What can one say from a partition that separates the graph into two humongous sets of nodes? In this work, we consider the problem of finding what we call cut projections. Given a (possibly small) subset of nodes from the graph, the objective is to partition the set of nodes such that this projects onto an induced cut of small conductance on the entire graph. The hope is that the bounds now strongly depend on the size of the specified set of nodes and only weakly on the size of the entire graph. We show how such cut projections can be obtained on regular graphs when the subset of nodes is chosen at random. While our theorems do not hold when these restrictions are omitted, the technique itself is likely to work well in practice.

In the distributed computing model, we again focus on the problem of performing random walks efficiently. Our results on this model work only on undirected networks (which is usually the case, since the communication paths are bidirectional). In

the first work, we show how to perform a random walk of length ℓ in $O(\ell^{2/3}D^{1/3})$ rounds; here D is the diameter of the network. This is a deterministic algorithm and we extend it to show how several random walks of length ℓ may be obtained efficiently. Specifically, K independent random walks of length ℓ can be performed in $O((K\ell)^{2/3}D^{1/3})$ rounds. Notice that for small diameter networks, this is a significant improvement over the naive and widely used algorithm that can perform each random walk of length ℓ in $O(\ell)$ rounds. Most peer to peer networks strive to maintain small diameters, and in such settings, this work provides a significant improvement for several applications.

In follow up work under the same congest model of distributed networks, we further improve the algorithm for performing several random walks. Specifically, we show a randomized algorithm that can perform one random walk in $O(\sqrt{\ell D})$ rounds w.h.p. or K random walks in $O(\sqrt{K\ell D})$ rounds w.h.p. Further, we show a lower bound that suggests that a single random walk requires $\Omega(\sqrt{\ell/\log \ell})$ rounds. Therefore, barring the diameter term, our approach for performing a single random walk is near-optimal. Our algorithm also introduces several new techniques and random walk properties that may be of independent interest. Further, we show how our algorithms for performing random walks efficiently can be used as subroutines for efficient distributed algorithms for two key properties: random spanning trees, and estimating mixing times. We show the fastest known distributed algorithms for both sampling a random spanning tree (which is useful in applications such as routing and generating sparsifiers), and estimating mixing times (which is crucial to understanding the connectivity of the network, a primary concern in peer to peer systems).

Finally, in the online framework of sketch based algorithms, we study the problem of estimating graph distances efficiently. Our objective is to be able to answer distance queries between a pair of nodes in real time. Since the standard shortest path

algorithms are expensive, our approach moves the time-consuming shortest-path computation offline, and at query time only looks up precomputed values and performs simple and fast computations on these precomputed values. More specifically, during the offline phase we compute and store a small sketch for each node in the graph, and at query-time we look up the sketches of the source and destination nodes and perform a simple computation using these two sketches to estimate the distance. Our algorithm is a simplification of the best algorithm for this problem by Thorup and Zwick [137] and matches its theoretical approximation guarantees: for undirected graphs with n vertices and $c > 1$, we prove that this simple technique gives a $2c - 1$ approximation to the actual distance using sketches of size $\tilde{O}(n^{1/c})$ per node. Further, unlike previous works, we also extend our algorithm to directed graphs and evaluate its performance on large graphs. Empirically, we show that on a large web graph, our algorithm gives very good estimates for both directed and undirected distances measured as the number of links between two nodes - the distances are accurate to within a small additive error.

In another work, we consider a variant of the streaming model that we call Best-order streaming model. This has a close relationship to many models of computation in other areas such as data streams, communication complexity, and proof checking and could be used in applications such as cloud computing. In this work we focus on graph problems where the input is a sequence of edges and the proofs is simply a reordering of the input edges. The verifier specifies the order in which the edges must be sent but the crucial point is that the prover may lie and the verifier needs to be able to check carefully. The goal is to see if graph problems can be solved much more efficiently in this model as compared to the traditional streaming model. We show that checking if a graph has a perfect matching is impossible to do deterministically using small (logarithmic) space. To contrast this, we show that randomized verifiers are powerful enough to check whether a graph has a perfect matching or is connected.

CHAPTER I

INTRODUCTION

Large Graphs. The scale of the web has greatly increased the need for efficient algorithms to process the underlying web-graph. Large graphs have become a common tool for representing real world data. We now routinely interact with search engines and social networking sites that make use of large web graphs and social networks behind the scenes. Many of these interactions happen in real time where users make some kind of a request or a query that needs to be serviced almost instantaneously. This thesis deals with algorithms for such large graphs. Problems in three specific models are addressed (1) Streaming algorithms, (2) Distributed (Peer-to-peer) network algorithms, and (3) Online algorithms.

Streaming algorithms - The basic premise of this model is that the graph can be stored in secondary storage while processing of this data needs to be performed using physical memory (RAM) which is a limited resource. The data is presented as a stream and any computation on the stream relies on using a small amount of memory. Many streaming algorithms exist for computing frequency moments (with matching lower bounds), quantiles, and norms [7, 87, 27, 26, 117, 78, 79], and computations over graphs including counting triangles, properties of degree sequences, and connectivity [18, 46, 53, 65, 91, 120]. While the basic requirements of streaming algorithms include small space and a small number of passes, these quantities can vary significantly from algorithm to algorithm. Several other variants of this model have also been considered that study different stream orders in which the input is presented - such as adversarial/arbitrary, randomized, or perhaps a more special (for e.g. sorted) order.

Distributed network algorithms - This model is extensively used for the study of graphs such as peer to peer networks, for e.g. the internet topology network. Without loss of generality, we assume that the graph is connected. Each node has a unique identifier and at the beginning of the computation, each node v accepts as input its own identifier. The nodes are allowed to communicate (only) through the edges of the graph G. We assume that the communication is synchronous and occurs in discrete rounds (time steps). We assume the $\mathcal{CONGEST}$ communication model, a widely used standard model to study distributed algorithms [131, 128]: a node v can send an arbitrary message of size at most $O(\log n)$ through an edge per time step. The goal is to minimize the number rounds required. Many fundamental network problems such as minimum spanning tree, shortest paths, etc. have been addressed in this model (e.g., see [115, 131, 128]). Such algorithms can be useful for large-scale resource-constrained and dynamic networks where running time is crucial.

Online Algorithms -There are several graphs online where users issue queries on a daily basis and expect quick responses, such as the web graph and social networks. Since user interactions should have low latency, one method to handle expensive operations is to do them offline as a precomputation and store the data so that they can be obtained quickly when required for a real-time operation. For example, PageRank consists of an expensive eigenvector computation over the web graph that can be performed offline and a simple online lookup of the PageRank score for each result. One standard approach is to perform a one-time offline computation. During this preprocessing phase, a certain auxiliary information is stored, which is sometimes a succinct structure associated with each node referred to as a *sketch*. At query time, the sketches of certain relevant nodes are retrieved to perform the desired computation exactly or approximately. Computing sketches offline for a large collection of objects to facilitate online computations has been studied extensively and is also used in many applications. For example, several search engines compute a small sketch of

documents to detect near-duplicate documents at query time; see [34, 58, 33] and references therein. This eliminates the need to compare large documents, which is time consuming. Instead it is achieved by comparing short sketches of these documents and measuring the similarity between these sketches.

Graph problems in this thesis - One of the main problems considered in this thesis is performing random walks efficiently on large graphs. Random walks play a central role in computer science, spanning a wide range of areas in both theory and practice, including web algorithms and distributed computing. Algorithms in many different applications use random walks as an integral subroutine. Random walks have played a key role in several applications extensively in the past few decades. In this thesis, we develop efficient algorithms for performing random walks on graph streams as well as distributed networks. These techniques are then extended to compute or estimate several related quantities such as PageRank (steady state) distributions, graph cuts and cut projections, mixing times, conductance of graphs, and random spanning trees. While random walks and their applications remain the main focus of this thesis, several other graph properties are also considered for some of the specific models, such as graph connectivity, undirected and directed distances etc.

1.1 Problems and Results

1.1.1 Streaming Algorithms for Random Walks and Applications

The naive technique for performing a random walk of length ℓ on a graph presented as a stream requires $\Theta(\ell)$ passes. We present an algorithm that can sample from a random walk of length ℓ in $O(\sqrt{\ell})$ passes, and $O(n)$ space. Further, we show how to generalize these bounds to a trade-off between passes and space given by $O(\sqrt{\ell/\alpha})$ passes, and $O(n\alpha)$ space, for any parameter $\alpha < 1$. We then show how several $O(n/\ell)$ independent random walks can be performed without increasing the space or passes.

Using these techniques for performing random walks efficiently, we show how one can estimate PageRank distributions, and graph partitions also on graph streams.

PageRank. In this study we compute the PageRank [31] of a large graph presented as a stream of edges in no particular order; neither is it required that all the edges incident on a vertex be grouped together in the stream. Real world networks such as the web and social networks can be modeled as large graphs. Other instances of large graphs include click graphs generated from search engine query logs, document-term graphs computed from large collection of documents etc. These graphs described above readily admit a streaming model. Moreover, they also admit link-based ranking algorithms like the PageRank algorithm to compute the relative importance of nodes in the graph.

Besides PageRank, other graph properties of interest include connectedness, conductance, mixing time, and the sparsest cut. It is well known and was first shown by Jerrum and Sinclair [90] that the mixing time M and conductance ϕ are related as $\phi^2/2 \leq \frac{1}{M} \leq 2\phi$ [1].

Cuts. The problem of finding sparse cuts on graphs has been studied extensively [25, 22, 92, 13, 136, 118]. Sparse cuts form an important tool for analyzing and partitioning large real world graphs, such as the web graph, click graphs from search engine query logs and online social communities [29]. For example, while the web graph may consist of several billions of nodes, in a given context, one may only be interested in the most important nodes such as those with high PageRank or those nodes (representing web pages) relevant to a specific search query. Specifically, we may be interested in finding how connected components of the graph partition these nodes, or we may wish to compute the diameter of the graph with respect to these

[1] Note that mixing time is well-defined only for *aperiodic* graphs. Bipartite graphs, for example, are not aperiodic. On the other hand, the bipartiteness of a graph can be checked in a single pass, see e.g., [65]

nodes, or perhaps compute the distance between these nodes with respect to the original graph. Such operations not only enable us to understand the structure of a facet of the graph, but also make it feasible to visualize the graph using a much smaller set

In this work, we present a streaming algorithm for finding how a sparse cut partitions a small random set of nodes when the graph is presented as a stream of edges in no particular order. Our streaming algorithm uses space sublinear in the number of nodes. We also provide an algorithm for finding a sparse cut on the entire graph.

1.1.2 Distributed Algorithms for Random Walks and Applications

Applications of random walks in networks include token management [89, 23, 44], load balancing [93], small-world routing [101], search [143, 2, 42, 72, 114], information propagation and gathering [24, 97], network topology construction [72, 105, 111], checking expander [56], constructing random spanning trees [32, 17, 16], monitoring overlays [124], group communication in ad-hoc network [57], gathering and dissemination of information over a network [6], distributed construction of expander networks [105], and peer-to-peer membership management [68, 144].

Random walks have also been used to provide uniform and efficient solutions to distributed control of dynamic networks [35]. The paper of [143] describes a broad range of network applications that can benefit from random walks in dynamic and decentralized settings. For further references on applications of random walks to distributed computing, see, e.g. [35, 143]. A key purpose of random walks in many of these network applications is to perform node sampling. Random walk-based sampling is simple, local, and robust. While the sampling requirements in different applications vary, whenever a true sample is required from a random walk of certain steps, all applications perform the walks naively. In this work we present the first non-trivial distributed random walk sampling algorithms in arbitrary networks that are significantly faster than the existing (naive) approaches.

Random walks are also very useful in providing uniform and efficient solutions to distributed control of dynamic networks [35, 143]. Random walks are local and lightweight and require little index or state maintenance which make them especially attractive to self-organizing dynamic networks such as Internet overlay and ad hoc wireless networks.

Although using random walks help in improving the performance of many distributed algorithms, all known algorithms perform random walks naively: Each walk of length ℓ is performed by sending a token for ℓ steps, picking a random neighbor with each step. Is there a faster way to perform a random walk distributively? We present an algorithm that performs a random walk in $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds with high probability, where D is the diameter of the network. We further extend these techniques to perform multiple random walks.

In subsequent work, we present a randomized algorithm that runs in time $\tilde{O}(\sqrt{\ell D})$ rounds w.h.p. We then present an almost matching lower bound that applies to a general class of distributed algorithms (our algorithm also falls in this class). Finally, we present two key applications of our algorithm. The first is a fast distributed algorithm for computing a random spanning tree, a fundamental spanning tree problem that has been studied widely in the classical setting (see e.g., [95] and references therein). To the best of our knowledge, our algorithm gives the fastest known running time in an arbitrary network. The second is to devising efficient decentralized algorithms for computing key global metrics of the underlying network — mixing time, spectral gap, and conductance. Such algorithms can be useful building blocks in the design of *topologically (self-)aware* networks, i.e., networks that can monitor and regulate themselves in a decentralized fashion. For example, efficiently computing the mixing time or the spectral gap, allows the network to monitor connectivity and expansion properties of the network.

1.1.3 Other: Online and Best-Order Streaming Algorithms

1.1.3.1 Sketch-based Online Distance Oracles

One fundamental operation on large graphs is finding shortest paths between a pair of nodes. This problem is not only a common building block in many algorithms, but is also a meaningful operation in its own right. For example, in a social network one may be interested in finding the shortest sequence of friends that connects one to a celebrity. However, given the large size of the various web graphs, shortest-path computation is challenging. It is not feasible to store a web-scale graph in the main memory of a single machine. Moreover, running Dijkstra's well known shortest-path algorithm [45] on a web graph containing tens of billions of nodes and trillions of edges would take several hours, if not days. Even in a distributed setting, if the computation is parallelized, the sequentially dependent nature of Dijkstra's computation would require huge amounts of communication. Furthermore, in a real-time computation, only a small amount of resources – memory access, CPU cycles – are available for a single shortest distance query. As we would like to do this in real time with minimal latency, it becomes important to use small amounts of resources per distance query. We study the problem of estimating distances between two nodes online in such large graphs.

One approach, the online framework we consider, is to perform a one-time offline computation. Subsequently, the stored space post the offline computation phase is used while answering online queries. As the pre-computed information, our algorithms store some auxiliary information with each node that can facilitate a quick distance computation online in real time. This auxiliary information is then used in the online computation that is performed for every request or query. One can view this auxiliary information as a *sketch* of the neighborhood structure of a node that is stored with each node. Simply retrieving the sketches of the two nodes should be sufficient to estimate the distance between them. The sketches need to be small, and there needs

to be an algorithm that can quickly and easily compute or approximate the distance between query nodes from their corresponding sketches. As the computation of the sketches is an offline computation, one can afford to spend more resources on this one-time preprocessing. We present such a sketch based algorithm that can be viewed as a simplification of the seminal work of Thorup-Zwick [137] for undirected graphs. We achieve the same theoretical guarantees as them, further our algorithms and analysis are significantly simpler. We also extend our algorithms to directed graphs, without proof. Finally, we perform extensive experiments for undirected as well as directed distances on a crawl of the web graph. Our algorithms perform extremely well (much better than the proved theoretical guarantee) and significantly outperform some previous approaches.

1.1.3.2 Best-Order Streaming Model

This work is motivated by cloud computing and streaming algorithms. We wish to answer the question of how efficient can space restricted streaming algorithms be.

Many big companies such as Amazon [1] and salesforce.com are currently offering *cloud computing* services. These services allow their users to use the companies' powerful resources for a short period of time, over the Internet. They also provide some softwares that help the users who may not have knowledge of, expertise in, or control over the technology infrastructure ("in the cloud") that supports them.[2] These services are very helpful, for example, when a user wants a massive computation over a short period of time.

Now, let's say that you want the cloud computer to do a simple task such as checking if a massive graph is strongly connected. Suppose that the cloud computer gets back to you with an answer "Yes" suggesting that the graph is strongly connected. What do you make of this? What if there is a bug in the code, or what if there was

[2]http://www.ebizq.net/blogs/saasweek/2008/03/distinguishing_cloud_computing/.

some communication error? Ideally one would like a way for the cloud to *prove* to you that the answer is correct. This proof might be long due to the massive input data; hence, it is impossible to keep everything in your laptop's main memory. Therefore, it is more practical to read the proof as a *stream* with a small working memory. Moreover, the proof should not be too long – one ideal case is when the proof is the input itself (in different order). This is the model considered in this work.

Recall that the basic premise of streaming algorithms is that one is dealing with a humongous data set, too large to process in main memory. The algorithm has only sequential access to the input data; this called a *stream*. In certain settings, it is acceptable to allow the algorithm to perform multiple passes over the stream. However, for many applications, it is not feasible to perform more than a single pass. The general streaming algorithms framework has been studied extensively since the seminal work of Alon, Matias, Szegedy [7].

Models diverge in the assumptions made about what order the algorithm can access the input elements and several variants have been considered. The most stringent restriction on the algorithm is to assume that the input sequence is presented to the algorithm in an adversarial order. A slightly more relaxed setting, that has also been widely studied is where the input is assumed to be presented in randomized order [38, 79, 80]. Aggarwal et. al. [3] proposed that if the algorithm has a power to sort the stream in one pass then it is easier to solve some graph problems (although not in one or constant passes), and several other variants have been considered.

We ask the following fundamental question:

> *If the input is presented in the best order possible, can we solve problems efficiently?*

Intuitively, this means that the algorithm processing the stream can decide on a *rule* on the order in which the stream is presented. We call this the best-order stream model. We present upper and lower bounds for various graphs problems under this

model.

1.2 Organization

The work on performing random walks and estimating PageRank on graph streams is presented in Chapter 2. The subsequent work that uses these techniques for finding graph partitions is then presented in Chapter 5. The initial part of the work on performing random walks in distributed networks, is presented in Chapter 3. The subsequent and improvement algorithms, lower bounds, and applications for random walks in this model are detailed in Chapter 4. Chapter 6 then talks about the online sketch-based algorithms for distance computation on large graphs. Finally, work on the best-order streaming model is described in Chapter 7.

CHAPTER II

PERFORMING RANDOM WALKS, AND ESTIMATING PAGERANK ON GRAPH STREAMS

In this chapter, the study [48] focuses on computations on large graphs (e.g., the web-graph) where the edges of the graph are presented as a stream. The objective in the streaming model is to use small amount of memory (preferably sub-linear in the number of nodes n) and a smaller number of passes.

In the streaming model, we [48] show how to perform several graph computations including estimating the probability distribution after a random walk of length l, mixing time, and the conductance. We show how to estimate the mixing time, M, of a random walk and related quantities such as the conductance of the graph. A key ingredient of our approach is to perform random walks efficiently in the streaming model. By applying our algorithm for computing probability distribution on the web-graph, we can estimate the *PageRank* p of any node up to an additive error of $\sqrt{\epsilon p}+\epsilon$ in $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes and $\tilde{O}(\min(n\alpha+\frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}}+\frac{1}{\epsilon}M\alpha, \alpha n\sqrt{M\alpha}+\frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}}))$ space, for any $\alpha \in (0,1]$. In particular (for $\epsilon = M/n$, $\alpha = M^{-\frac{1}{2}}$), we can compute the approximate PageRank values in $\tilde{O}(nM^{-\frac{1}{4}})$ space and $\tilde{O}(M^{\frac{5}{8}})$ passes. In comparison, a standard implementation of the PageRank algorithm will take $O(n)$ space and $O(M)$ passes.

2.1 Related Work and Contributions

In this study we compute the PageRank [31] of a large graph presented as a stream of edges in no particular order. While the basic requirements of streaming algorithms include small space and a small number of passes, these quantities can vary significantly from algorithm to algorithm. After Henzinger et. al. [86] showed linear lower

bounds on the "space × passes" product for several problems including connectivity and shortest path problems, the work of Feigenbaum et. al. [65] studied the value of space in computing graph distances in the streaming model. Specifically, they studied approximate diameter and girth estimation. Their techniques are based on a single-pass randomized algorithm for constructing a $(2t+1)$-spanner. Demetrescu et. al. [53] proposed streaming graph algorithms using sublinear space and passes to compute single-source shortest paths on directed graphs and s-t connectivity on undirected graphs. In this study we propose algorithms that require sublinear space and passes to compute the approximate PageRank values of nodes in a large directed graph.

We now give a brief description of PageRank. Given a web-graph representing the web pages and links between them, PageRank [31] computes a query-independent score for the web pages, taking into account endorsement of a web page by other web pages. The endorsement of a web page is computed from the in-links pointing to the page from other web pages. Alternately, the PageRank algorithm can also be viewed as computing the probability distribution of a random surfer visiting a page on the web. In general the PageRank of a page u is dependent on the PageRank of all pages v that link to u as $PR(u) = \sum_{(v,u)\in E} PR(v)/d(v)$ where $d(.)$ represents the out-degree. A standard implementation of the algorithm requires several iterations, say M, before the values converge. Alternately, this process can be viewed as a random walk requiring M steps. The typical length is about 200. In fact, the random walk is performed on a slightly modified graph that captures a random reset step in the PageRank algorithm. This step was introduced to model the random jump of a surfer from a page with no out-links to another page on the web and it also improves the convergence time of the algorithm. Typically, in a PageRank computation, the nodes with large PageRank are of interest.

2.1.1 Contributions of this study

A random walk of length l can be modeled as a matrix-vector computation $v = u.A^l$, where u (a row vector) is the initial distribution on the nodes, A is the transition matrix that corresponds to a single step in the random walk on the underlying graph, and v is the final distribution after performing the walk. The problem of computing a single destination of a random walk of length l starting from a node picked from the distribution u is same as sampling a node from the distribution v. So by simply maintaining an array of size n that represents the probability distribution, we can compute v in l passes and $O(n)$ space. Thus, a standard implementation of the PageRank algorithm, that computes the stationary distribution, will require M passes and $O(n)$ space, where M is the mixing time. In comparison, our work requires $\tilde{O}(\sqrt{M})$ passes and $o(n)$ space to compute the PageRank of nodes with values greater than M/n. This requires the knowledge of the mixing time. We also provide an algorithm to estimate the mixing time. In this work we provide algorithms on a graph stream for the following problems -

- Running a single random walk of length l in $\tilde{O}(\sqrt{l})$ passes. In fact, we show how to perform n/l independent random walks using space sublinear in n and passes sublinear in l using the single random walk result as a subroutine,
- Using the sampled random walks obtained in the previous result, we show how to compute an approximate probability distribution (the PageRank vector) of nodes after a random walk of length l.
- Finally, using probability distributions of random walks for different lengths, we show how to estimate the mixing time, which in turn gives an estimate of the conductance of the graph.

For all these results, the main goal is to use as few passes over the stream as possible, while using space sub-linear in the number of nodes in the graph. Notice

that for a dense graph (number of edges $\Omega(n^2)$), the space used by our algorithms are asymptotically less than square-root of the length of the stream. To compute the probability distribution over nodes after a random walk of length l, a naïve algorithm uses l passes and $O(n)$ space by performing l matrix-vector multiplications. We show how to approximate the same distribution; For any node with probability p in the distribution, we can approximate p within $p \pm \sqrt{\epsilon p} \pm \epsilon$ in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(\min(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + \frac{1}{\epsilon}l\alpha, \alpha n\sqrt{l\alpha} + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}}))$ space. Note that approximating p within $p \pm \sqrt{\epsilon p} \pm \epsilon$ can also be viewed as a $1 \pm \sqrt{\epsilon/p} \pm \epsilon/p$ approximation ratio which is close to 1 for p much larger than ϵ. This means we can estimate the probability value with high accuracy for nodes with large probability. Note that in the context of PageRank computation, we set l to the mixing time M of the random walk. For concreteness, this means we can estimate the PageRank p of a node within $p \pm \sqrt{\epsilon p} \pm \epsilon$ in $\tilde{O}(nM^{-\frac{1}{4}})$ space and $\tilde{O}(M^{\frac{3}{4}})$ passes for $\epsilon = M/n$. We also show how to find what we call the ϵ-near mixing time, i.e. the time taken for the probability distribution to reach within ϵ of the steady state distribution under the L_1-norm. This automatically gives a result for estimating the *conductance* of the graph.

2.1.2 Related Work

Data streaming algorithms became popular since the famous result of Alon et. al. [7] on approximating frequency moments. Since then, there has been a surge of papers looking at various problems in the data streams setting. In particular, there has been significant attention to computing various frequency moments as they provide important statistics about the data stream. Tight results are known for computing some of them, while others remain open [27, 87]. Recent streaming results include estimating earth-movers distance [88], communication problems [67, 141], counting triangles in graphs [18, 91, 36], quantile estimation [79, 80], sampling and entropy information [82, 81], and graph matchings [120].

This is by no means a comprehensive summary of all the results on data streams. There are many more studies on streaming problems including filtering irrelevant data from the stream, low rank approximation and fast multiplication of matrices, characterizing sketchable distances, scheduling problems, work on dynamic geometric problems, generating histograms, and finding long increasing subsequences in data streams.

In comparison to the work on aggregating statistics of a data stream, the work on graphs as data streams is limited. Demetrescu et.al. [53] show space-pass trade-offs for shortest path problems in graph streams. In general, it seems hard to approximate many properties on graphs while maintaining sub-linear space in the number of vertices in the graph, by performing only a constant passes over the stream.

A very interesting piece of work is due to Sarlos *et al* [134] who give approaches to finding summaries for hyperlink analysis and personalized PageRank computation. Their study is not under the data streams setting; rather, they use sketching techniques and construct simple deterministic summaries that are later used by the algorithms for computing the PageRank vectors. They give lower bounds to prove that the space required by their algorithms is *optimal* under their setting. Given an additive error of ϵ and the probability δ of an incorrect result, their disk usage bound is $O(n \log(1/\delta)/\epsilon)$.

There has also been work on s, t connectivity [63, 11] not under the streams setting; however it is not clear that extending to the streaming model is possible. Recent work by Wicks and Greenwald [139] shows an interesting approach to parallelizing the computation of PageRank. McSherry [121] exploits the link structure of the web graph within and across domains to accelerate the computation of PageRank. Andersen et. al. [10] computes local cuts near a specified vertex using personalized PageRank vectors

The main ingredient in all our algorithms is to perform random walks efficiently.

We begin by presenting the algorithm for running one random walk of length l in a small number of passes in Section 2.3. The main idea in this algorithm is to sample each of the n nodes independently with probability α and perform short (length w) random walks from each of the sampled nodes in w passes. We then try to *merge* these walks to form longer random walks from the source. The main idea in estimating the probability distribution or mixing time is running several such random walks. However, running several random walks requires some additional ideas to ensure small space. We describe how to efficiently run a large number of random walks in Section 2.4. This also gives an algorithm for approximating the probability distribution after a random walk. The algorithm for approximating the mixing time uses these ideas and is described in Sections 2.5. Section 2.6 provides an alternate algorithm for estimating the probability distribution with higher accuracy more efficiently for certain values of ϵ.

2.2 Primer on Random Walks and PageRank

Random Walks. Given a weighted directed graph G, the random walk process is defined as follows. The probability of transitioning from a node i to a node j, for $i, j \in V(G)$ and $(i, j) \in E(G)$ is equal to $\frac{w_{ij}}{\sum_{(i,k)\in E(G)} w_{ik}}$ where w_{ij} is the nonnegative weight on the edge (i, j). For $(i, j) \notin E(G)$, the probability of transitioning from i to j is 0. So, the probability of transitioning from a node to a neighbor is proportional to the weight of the corresponding edge. For undirected graphs, the weight is equal in both directions.

The random walk process can be defined for undirected graphs as well, where the sum above is taken over all undirected edges $(i, k) \in E(G)$. For unweighted graphs, the probability of transitioning from a vertex i to any neighbor is the same, and is equal to $1/d(i)$ where $d(i)$ is the degree of the vertex i in G.

Transition Matrix. The random walk transition probabilities on a graph is often expressed in the form a matrix, called the transition matrix A. The (i, j)-th entry in the matrix denotes the probability of transition from i to j by the random walk process. The transition matrix A can easily be obtained from the adjacency matrix of the graph. Every row sum of the matrix is 1.

Therefore, for edges (i, j) in the graph we have,

$$A(i,j) = \frac{w_{ij}}{\sum_{(i,k)\in E(G)} w_{ik}}$$

For all other pairs (i, j), $A(i, j) = 0$. Starting from a probability distribution u (row-vector) over the nodes, the distribution after l steps of the random walk can be computed as $u.A^l$.

Steady State Distribution. The steady state distribution of a random walk process is defined as the distribution that the walk approaches, as the number of steps in the walk goes to infinity. The steady state distribution vector v satisfies $v = v.A$. The steady state vector is uniquely determined under some basic conditions [90] such as the graph being connected (or strongly connected in the case of directed graphs) and aperiodic. For undirected graphs, the vector v is determined by the degree distribution, and for node i, v_i is equal to the $\frac{deg(i)}{2m}$ where $deg(i)$ is the degree of node i and m is the total number of edges.

PageRank. In real world graph applications, the PageRank vector is the steady state distribution of a random walk that is a slight modification of the standard random walk over the graph. At any step, the random walk process defined previously is followed with probability $\beta \leq 1$. With probability $1 - \beta$, node i transitions to a randomly chosen node. In particular, the transition matrix with random resets incorporated is as follows.

$$A(i,j) \;=\; \beta \cdot \frac{w_{ij}}{\sum_{(i,k)\in E(G)} w_{ik}} + (1-\beta)\frac{1}{n}$$

Alternatively one can think of this as modifying the graph by adding a clique with weight β to the given graph (weighted by $1-\beta$). Here, n is the number of vertices in the graph. One of the reasons these random reset edges are incorporated is to force the graph to become strongly connected. This way, the PageRank vector becomes independent of the initial distribution. Since these edges can be implicitly assumed to be appearing in the stream by the algorithm, here after whenever we mention a graph or a random walk, we assume that it is over this modified graph with random reset edges included.

We will provide a streaming algorithm to estimate the steady state distribution of a random walk on a graph; since the PageRank vector is the steady state distribution of a random walk on a slightly altered graph, we can still use our algorithm by implicitly handling the edges corresponding to the random resets at run time.

Mixing Time and Conductance. The number of steps of the random walk process that lead any starting distribution to close to the steady state distribution is called the mixing time of the graph. Therefore, if M is the mixing time, then for any starting distribution u, the transition matrix A of the graph satisfies $u.A^M$ is close to $u.A^{M+1}$. Formally, the mixing time is defined as the smallest M such that the variation distance $|u.A^M - u.A^{M+1}|_1$ is at most ϵ for all u. Here ϵ is normally chosen to be a constant $1/2e$.

The conductance of a graph G is defined as $\phi(G) = \min_{S:|S|\leq|V|/2} \frac{E(S,V(G)\setminus S)}{E(S)}$ where $E(S, V(G) \setminus S)$ is the number of the edges spanning the cut $(S, V(G) \setminus S)$ and $E(S)$ is the number of edges on the smaller side of the cut. Conductance is a good measure of how separable a graph is into two pieces. The conductance Φ of a graph and the

mixing time M are related by $\Theta(1/M) \leq \Phi \leq \Theta(1/\sqrt{M})$ as shown in [90].

2.3 Single Random Walk

We first present an algorithm to perform a single random walk over a graph stream efficiently. The naïve approach is to do this in $O(1)$ space and l passes by performing one step of the random walk with every pass over the stream. At the other extreme, one can perform a random walk of length l in 1 pass and $O(nl)$ space by sampling l edges out of each of the n nodes in one pass. Subsequently, with these nl edges stored, it is possible to perform a random walk of length l without any more passes, as with l edges out of each node, the random walk cannot get stuck at a node before completing a walk of length l. In this section, we show the following result.

Theorem 2.3.1. *One can perform a random walk of length l in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes* with high probability[1] *and $\tilde{O}(n\alpha + \sqrt{\frac{l}{\alpha}})$ space, for any choice of α with $0 < \alpha \leq 1$.*

Setting $\alpha = 1$, we get the following corollary.

Corollary 2.3.2. *One can perform a random walk of length l in $\tilde{O}(\sqrt{l})$ passes and $\tilde{O}(n)$ space.*

We start by describing the overall approach of our algorithm.

Perform short random walks out of sampled nodes - The main idea in our algorithm is to sample each node with probability α independently and perform short random walks of length w from each sampled node; this is done in w passes over the stream. The algorithm tries to extend the walk from the source by *merging* these short walks to form a longer random walk. It may get stuck in one of two ways. First, the walk may end up at a node that has not been sampled. Second, the walk may end up at one of the sampled nodes for a second time; notice that its stored w

[1] *also abbreviated as w.h.p. The probability is at least $1 - \frac{1}{poly(nl)}$ if we include $\log(nl)$ factors in the $\tilde{O}$. All our theorems hold w.h.p.*

length walk cannot be used more than once in order to preserve the randomness of the walk. Note that sampling each node with probability α can be done by using a pseudo-random hash function on the node id.

Handling stuck nodes - While constructing the walk if it gets stuck at a node, from which no unused w-length walk is available, we will refer to such a node as a *stuck* node. We handle stuck nodes as follows. We keep track of the set S of sampled nodes whose w length walks have already been used in extending the random walk so far. We sample s edges out of the stuck node and each node in S in one pass. (If the degree of a node is present in the stream it can be done in one pass; otherwise one can do this in two passes, one to compute its degree and another to sample the edges.)

We then extend the walk as far as possible using these newly sampled edges. If the new end-point is a sampled node whose w-length walk has not been used (i.e., it is not in S), then we continue merging as before. Otherwise, if the new end-point is a new stuck node, we repeat the process of sampling s edges out of S and all the stuck nodes visited since the last w-length walk was used. Finally, if the new end-point is not a stuck node, we continue appending w length walks as before.

We need to argue that whenever the algorithm hits a stuck node, it makes sufficient progress with each pass. Note that after each round of sampling s edges out of the stuck nodes and the nodes in S, the walk is extended further. Either the walk reaches a node that is not stuck, thereby resulting in w progress, or the walk is extended until it again reaches a node that is stuck. Notice that in the latter case, s steps of progress is made. The point is that the we cannot keep finding new stuck nodes repeatedly for too long as each new node is a sampled node with probability α. So, it is unlikely we will visit more than $\tilde{O}(1/\alpha)$ new stuck nodes in a sequence before becoming unstuck. These steps are detailed in the algorithm SingleRandomWalk.

Algorithm 1 SINGLERANDOMWALK(u, l)

Input: Starting node u, and desired walk length l.
Output: $\mathcal{L}_u$ the random walk from u of length l.
$T \leftarrow$ set of nodes obtained by sampling each node independently with probability α (in one pass).
In w passes, perform walks of length w from every node in T. Let $W[t] \leftarrow$ the end point of the walk of length w from $t \in T$ (the nodes in T whose w length walks get used towards $\mathcal{L}_u$ will get included in S).
$S \leftarrow \{\}$(we will refer to the nodes in S as *centers*).
Initialize $\mathcal{L}_u$ to a zero length walk starting at u. Let $x \leftarrow u$.
while $|\mathcal{L}_u| < l$ **do**
    1. if ($x \in T$ and $x \notin S$) extend $\mathcal{L}_u$ by appending the walk (implicit in) $W[x]$. $S \leftarrow S \cup \{x\}$. $x \leftarrow W[x]$, the new end point of $\mathcal{L}_u$. {this means we have a w length walk starting at x that has not been used so far in $\mathcal{L}_u$}

    2. if ($x \notin T$ or $x \in S$) HANDLESTUCKNODE(x, T, S, $\mathcal{L}_u$, l). {this means that either x was not in the initial set of sampled nodes, or x's w-length walk has already been used up}
end while

The notation in the algorithm SINGLERANDOMWALK uses T to denote the sampled nodes obtained by sampling each node with probability α independently. The table W indexed by a sampled node (say t) stores the end point of the w length walks starting at t as $W[t]$. Note that this table can be populated in w passes while using $O(\alpha n)$ space. The set S keeps track of all nodes in T whose w length walks get used up. The algorithm continues extending the walk using the w length walks implicitly stored in the table W until it finds a stuck node. The module HANDLESTUCKNODE proceeds by sampling s edges out of $S \cup R$ where R is the set of stuck nodes visited in the current invocation.

Remark 2.3.3. *The length of the walk produced by algorithm* SINGLERANDOMWALK *could exceed l slightly (by at most w). To prevent this we can run the algorithm till we get a walk of length at least $l - w$ and then extend this walk to length l in at most w additional passes.*

We begin the analysis with a lemma that follows immediately from the algorithm.

Algorithm 2 HANDLESTUCKNODE(x, T, S, $\mathcal{L}_u$, l)

1: $R \leftarrow x$.
2: **while** $|\mathcal{L}_u| < l$ **do**
3: $E \leftarrow$ sample s edges (with repetition) out of each node in $S \cup R$.
4: Extend $\mathcal{L}_u$ as far as possible by walking along the sampled edges in E (on visiting a node in $S \cup R$ for the k-th time, use the k-th edge of the s sampled edges from that node).
5: $x \leftarrow$ new end point of $\mathcal{L}_u$ after the extension. One of the following cases arise.
   1. if ($x \in S \cup R$) **continue** {no new node is seen, at least s progress has been made.}
   2. if ($x \in T$ and $x \notin S \cup R$) **return** {this means that x is a node that has not been seen in the walk so far, and x was among the set of nodes sampled initially; therefore, the w-length walk from x has not been used}
   3. if ($x \notin T$ and $x \notin S \cup R$) $R \leftarrow R \cup \{x\}$. {this means that x is a new node that has not been visited in this invocation, and x is not in the initial set sampled nodes T}
6: **end while**

Lemma 2.3.4. $|S| \leq \frac{l}{w}$.

Proof. A node is added to the set S only after we use a w length walk from one of the sampled nodes. If we perform a walk of length l, we will end up using at most $\frac{l}{w}$ walks of length w. □

We now state and prove the main claim that is needed to bound the number of passes required by algorithm SINGLERANDOMWALK to perform a random walk of length l.

Claim 2.3.5. *With every additional pass over the edge stream (after the first w passes), the length of the random walk $\mathcal{L}_u$ either increases by s, or if it does not increase by s then with probability α it increases by w. Further $|R| \leq \tilde{O}(1/\alpha)$ with high probability.*

Proof. We only need to examine the algorithm HANDLESTUCKNODE. An additional pass over the stream is made when s edges are sampled from every node in $S \cup R$. This happens when the algorithm gets stuck at a new stuck node in R. After a pass

over the stream, either the algorithm makes s progress, or a *new node* is visited. This is because, after a pass, all nodes that were stuck (i.e. nodes whose w-length walks are used up) now have s edges sampled out of them. If on extending the walk, the walk continues to visit these nodes, the algorithm completes at least s steps before getting stuck again at one of these nodes. However, the algorithm may exit the set of stuck nodes and end up at a node outside, before completing s steps. In this case, with probability α, the new node is in T (since T contains each node with probability α), and with probability $1-\alpha$, it is a new stuck node. This is because each of these new nodes was sampled independently with probability α at the beginning of the algorithm. If the new node is not a stuck node, w progress is made. The probability of not seeing a new node in T is $1-\alpha$ with every additional pass that lands at a new node. Therefore, the probability that more than $O(\log(nl)/\alpha)$ new stuck nodes are seen before a new node in T is seen is small (at most $1/poly(nl)$) by Chernoff Bounds. So w.h.p., $|R|$ is less than $\tilde{O}(1/\alpha)$ in each invocation of HandleStuckNode. □

We are now ready to prove Theorem 2.3.1.

Proof of Theorem 2.3.1. **Correctness:** We first argue that the walk of length l from source u generated by our algorithm is indeed a *random* walk. Notice that the algorithm uses each w-length walk only once in the walk $\mathcal{L}_u$. Note that the algorithm never reuses any randomly sampled edges or walks. Whenever we sample s edges, we pick the ith sampled edge, when visiting the node for the ith time. Therefore randomness is maintained. It is important to note that while sampling s edges, we (correctly) allow the same edge to be sampled multiple times; in particular, this would definitely happen for a vertex with degree less than s.

Space: We need space $O(\alpha n)$ for storing the sampled nodes and the end point of their w length walks. Using Lemma 2.3.4, sampling s edges from every node in S requires $O(s\frac{l}{w})$ space, while sampling s edges from the nodes not in R takes up an

additional $\tilde{O}(s\frac{1}{\alpha})$ space as $|R| = \tilde{O}(1/\alpha)$ w.h.p. $\tilde{O}(n\alpha + s(\frac{l}{w} + \frac{1}{\alpha}))$.

Passes: The most crucial observation in analyzing the number of passes required by the algorithm SingleRandomWalk is Claim 2.3.5. This claim states that in case s progress is not made in a pass then with probability α w progress is made. The number of passes in which s progress is made is at most l/s. Now let us bound the number of passes in which case s progress is not made – then with probability α, w progress is made which can happen at most l/w times. Now, in $\tilde{O}(1/\alpha)$ such passes at least once w progress is made. Thus the number of passes when s progress is not made is at most $\tilde{O}(\frac{l}{w\alpha})$ w.h.p (the $\tilde{O}$ includes $\log(nl)$ factors to achieve a high probability of $1 - 1/poly(nl)$). Additionally, w passes are used for generating the w length walks from each of the sampled $O(\alpha n)$ nodes.

Therefore, the total number of passes used in the algorithm is $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$ with high probability. Setting $s = \sqrt{l\alpha}$ and $w = \sqrt{\frac{l}{\alpha}}$ completes the proof. □

Note that SingleRandomWalk takes sublinear space and passes even for performing very long ($O(n)$ length) random walks. Setting $l = O(n)$ and choosing $\alpha = n^{-\frac{1}{3}}$ in Theorem 2.3.1 gives the following corollary.

Corollary 2.3.6. *One can perform a random walk of length $O(n)$ in $\tilde{O}(n^{2/3})$ passes and $\tilde{O}(n^{2/3})$ space.*

The above algorithm can easily be extended to the case when the starting node of the random walk comes from a distribution, rather than a specific node. In this case, one can sample a node from the initial distribution and use this as the source node for the random walk.

Notice that if we wanted to perform a larger number of independent random walks using this algorithm directly, the space required would increase linearly in the number of walks, while the passes would remain unchanged. The bottle-neck in the space requirement would arise due to two reasons. First, the algorithm would need to store

multiple w-length walks from each sampled node, one for each random walk. Second, many of these random walks could get stuck at the same time, and the algorithm may be required to sample s edges out of the centers of many walks. In the following section, we reduce the space requirements arising in both these scenarios by trying to identify the *appropriate* number of w-length walks required for each sampled node.

The above algorithm only samples end points from the distribution at length l. One can in fact regenerate the entire walk by using a pseudo-random generator for the coin tosses during the algorithm.

Remark 2.3.7. *Note that since we only store the end-points of w length walks in W, the internal nodes are not available in $\mathcal{L}_u$ at the end of the algorithm* SingleRandomWalk. *These w-length walks can be reconstructed by making pseudo-random choices while creating the w length random walks in Step 3 of* SingleRandomWalk*, and reusing the coin tosses to reconstruct them at the end. A single pseudo-random hash function can be used to generate all the coin tosses.*

2.4 Estimating probability distribution by performing a large number of random walks

We now show how to estimate the probability distribution of the destination node after performing a random walk. We achieve this by performing several random walks. The source node may either be fixed or chosen from a certain initial distribution. A naïve method that uses algorithm SingleRandomWalk to perform K random walks would require $O(K(n\alpha + \sqrt{\frac{l}{\alpha}}))$ space. In this section, we show how algorithm SingleRandomWalk can be extended to perform n/l random walks without significant increase in the space-pass complexity. Specifically, we show the following result.

Theorem 2.4.1. *One can perform K random walks of length l in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(n\alpha + K\sqrt{\frac{l}{\alpha}} + Kl\alpha)$ space for any choice of α with $0 < \alpha \leq 1$.*

In particular, if $K = \frac{n}{l}$, then for $\alpha \geq l^{-1/3}$, the space requirement is $O(n\alpha)$ which is as good as the space complexity of SINGLERANDOMWALK. We first describe how Theorem 2.4.1 can be used to estimate the probability distribution after a random walk of length l.

By performing a large number of random walks and computing the fraction of walks that end at a given node gives us an estimate of the probability that a random walk ends at this node. If the actual probability of ending at a node is p, then by setting $K = \Theta(\frac{\log n}{\epsilon})$, we get an estimate for p with accuracy $p \pm \sqrt{\epsilon p} \pm \epsilon$. By Chernoff bounds, due to the $\log n$ factor, this estimate is valid w.h.p. for all nodes.

The specific form of the bound we require here (and use repeatedly in our analysis later on as well) is stated below as a lemma.

Lemma 2.4.2. *If the probability of an event X occurring is p, then in $t = \Theta(\log n/\epsilon)$ trials, the fraction of times the event X occurs is $(p \pm \sqrt{p\epsilon} \pm \epsilon)$ times w.h.p.*[2].

Proof. Given independent identically distributed random variables X_i, such that $Pr[X_i = 1] = p$ and $Pr[X_i = 0] = (1 - p)$, and t events $X_1, X_2, \ldots, X_t$, standard Chernoff or Hoeffding Bounds says for the lower tail $Pr[\frac{1}{t}\sum_{i=1}^{t} X_i < (1 - \delta)tp] < (\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}})^{tp} < e^{-tp\delta^2/2}$. For the upper tail, the bound is $Pr[\frac{1}{t}\sum_{i=1}^{t} X_i > (1 + \delta)tp] < (\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}})^{tp}$. Further, for $\delta > 2e - 1$, this bound reduces to $2^{-\delta tp}$ and for $\delta < 2e - 1$, this bound reduces to $e^{-tp\delta^2/4}$.

For the lemma, we choose $t = \Theta(\frac{\log n}{\epsilon})$, and $\delta = \sqrt{\frac{\epsilon}{p}} + \frac{\epsilon}{p}$. Now consider two cases, one where $p > \epsilon$ and when $p \leq \epsilon$. When $p \leq \epsilon$, the lower tail bound follows automatically since $p - \epsilon < 0$. For the upper tail bound, we have $\delta > 1$; in this case, the weaker bound is $2^{-\delta tp}$ and we show that this suffices to prove the lemma. We have $2^{-\delta tp} = 2^{-(\sqrt{\epsilon p}+\epsilon)t} \leq 2^{-\epsilon t} = 2^{-\log n} = 1/n$. Now consider the case when $p > \epsilon$. Now, $\delta < 2$ is small and so the lower and upper tail bounds are $e^{-tp\delta^2/2}$ and $e^{-tp\delta^2/4}$

[2] *probability* $1 - 1/n^{\Omega(1)}$

respectively. In this case, $\sqrt{\epsilon/p} > \epsilon/p$, therefore, we get an asymptotic bound of $e^{-\Theta(tp(\sqrt{\epsilon/p}^2)} = e^{-\Theta(\frac{\log n}{\epsilon}p(\epsilon/p))} = e^{-\Theta(\log n)} = \frac{1}{n^{\Theta(1)}}$. This completes the proof.

□

We then get the following corollary.

Corollary 2.4.3. *For any node with probability p in the probability distribution after a walk of length l, one can approximate its probability up to an accuracy of $p \pm \sqrt{p\epsilon} \pm \epsilon$ in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + \frac{1}{\epsilon}l\alpha)$ space for any choice of α with $0 < \alpha \leq 1$. This is a $(1 \pm \sqrt{\frac{\epsilon}{p}} \pm \frac{\epsilon}{p})$-factor approximation for a node with probability p in the distribution.*

Notice that for $p \gg \epsilon$, this is a constant factor approximation close to 1.

By applying this algorithm on the web-graph (including the random reset edges that are handling implicitly), we can estimate the PageRank vector up to an accuracy of $p \pm \sqrt{p\epsilon} \pm \epsilon$ for any node with probability p in the stationary distribution, in $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes and $\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}} + \frac{1}{\epsilon}M\alpha)$ space, where M is the mixing time of the graph. We will show later in Section 2.5 how to estimate the mixing time M of any graph. Note that our algorithm is also able to handle the random resets in the standard definition of PageRank by handling these transitions implicitly; that is the transition edges corresponding to the random resets can be included into the graph implicitly.

Our technique for performing a large number of walks uses algorithm SingleRandomWalk as a subroutine. The key idea in our approach is to *estimate* the probability r_i that the w length walk of node i is used in SingleRandomWalk. We then use the r_i's to store the appropriate number of w length walks from each sampled node for K executions of SingleRandomWalk. Estimating r_i however again requires performing random walks. For this purpose, we start by performing one random walk, then two, then four and so on, doubling the number of random walks in every phase, giving more and more accurate estimates of r_i for the sampled nodes.

Let us first define r_i for every node i assuming a given set of sampled nodes in Algorithm SingleRandomWalk. Note that this definition is dependent on the length of the walk but we omit write this as a superscript.

Definition 2.4.4 (r_i)**.** *For any sampled node i, define r_i to be the probability that on running the algorithm* SingleRandomWalk*, the w length walk of node i is used (and hence i gets included in the set of centers S in the performed walk of length l).*

Lemma 2.3.4 states that $|S| \leq l/w$. Notice that a node is included in S if and only if its w length random walk is used towards the random walk $\mathcal{L}_u$. By our definition of r_i, we have that $\sum_i r_i$ is equal to the expected size of $|S|$. From these two statements, we get the following observation.

Observation. $\sum_i r_i \leq \frac{l}{w}$.

We now describe the algorithm for performing a large number of random walks. Whenever we say *sample x walks of length w from i*, we mean take the end-points of x independent random walks of length w starting at i.

This algorithm runs in phases. To obtain K walks of length l, algorithm MultipleRandomWalk is run for $j = \log K$ phases. In phase $j+1$ we run $O(2^j \log n)$ parallel executions of SingleRandomWalk and use these to get an estimate $\tilde{r}_i$ of r_i to an additive error of $\sqrt{r_i/2^j} + 1/2^j$. This estimate is then used in the next phase to store the appropriate number of w length walks from each i. Note that, all the executions of SingleRandomWalk share the same set of sampled nodes.

We are now ready to prove Theorem 2.4.1.

Proof of Theorem 2.4.1. **Correctness:** We need to show that the number of w length

Algorithm 3 MULTIPLERANDOMWALK($\mathcal{I}$, l, K)

1: **Input**: Distribution of the source nodes, $\mathcal{I}$, and length of the walk, l
2: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
3: Perform phases 1 through $\log K$ as follows -
4: **Phase 1:**
   1. Perform $O(\log n)$ walks of length w from each of the sampled nodes, in w passes.
   2. Spawn $K_1 = O(\log n)$ instances of SINGLERANDOMWALK to obtain K_1 walks. All these instances use only the w-length walks in the previous step.
   3. For each sampled node $i \in T$, set estimate $\tilde{r}_i = n_i/K_1$ where n_i is the number of walks produced in the previous step that use i as a center.
5: **Phase** $(j+1)$**:** {The value of $\tilde{r}_i$ differs from r_i by an additive error of $\sqrt{r_i/2^j}+1/2^j$ w.h.p.}.
   1. In w passes, sample $O(2^j\tilde{r}_i \log n + \log n)$ random walks of length w for all nodes in T.
   2. Run $K_{j+1} = 2^j O(\log n)$ independent instances of SINGLERANDOMWALK using the w- length random walks sampled in the previous step.
   3. Set estimate $\tilde{r}_i = n_i/K_{j+1}$ where n_i is the number of walks that use i as a center.

walks we store from the sampled nodes in step 1 of each phase $j+1$ in MULTIPLERANDOMWALK is *sufficient* for K_{j+1} executions of SINGLERANDOMWALK. Note that (assuming previous phases have completed successfully) in each phase the estimate $\tilde{r}_i$ is accurate up to an additive error of $\sqrt{r_i/2^j}+1/2^j$ w.h.p; this follows from lemma 2.4.2 since we are using $K_j = \tilde{O}(2^j)$ walks to estimate r_i. Therefore, our estimated $\tilde{r}_i$ is w.h.p. at least $r_i - \sqrt{r_i/2^j} - 1/2^j \geq r_i/2 - 1/2^{j+1}$ (since $\sqrt{xy} \leq (x+y)/2$, for any positive reals x, y). It follows that the actual value r_i is at most $O(\tilde{r}_i + 1/2^j)$ w.h.p. The number of walks in phase $j+1$ that use this node as a center is w.h.p., by Chernoff bounds, at most $O(K_{j+1}r_i + \log n) \leq O(2^{j+1}\tilde{r}_i \log n + 2 + \log n) = O(2^j\tilde{r}_i \log n + \log n)$. This is exactly the number of w-length walks we sample in phase $j+1$. This implies that w.h.p. any time one of $K_{j+1} = 2^j O(\log n)$ walks hits a sampled node for the first time, there is an unused w length walk for that node, to extend it. By including $\log nKl$ factors in the $\tilde{O}$, each individual walk holds with probability at least

$1 - 1/poly(nKl)$; so it holds with high probability over all the K walks. No w length walk or sampled edge, is ever reused throughout the execution; therefore, all random choices in the eventual walk are independent.

Space:

Suppose we do t phases such that $2^t = K$. Total space required would include $O(n\alpha)$ space to store the sampled nodes. Further, to store the w length walks in phase t, we need $O(\sum_{i \in T}(2^t \tilde{r}_i \log n + \log n))$ space. Now $\tilde{r}_i \leq r_i + \sqrt{r_i/2^t} + 1/2^t \leq O(r_i + 1/2^j)$ since ($\sqrt{xy} \leq (x+y)/2$ for $x, y \geq 0$). So the space to store the w length walks in phase t is at most $O(\sum_{i \in T}(2^t r_i \log n + 1 + \log n)) = \tilde{O}(K(\frac{l}{w}) + n\alpha)$ space. And finally, the space for sampling s edges in each execution of HandleStuckNode amounts to $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s)$. Therefore, the algorithm MultipleRandomWalk uses a total space of $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s + \alpha n)$.

Passes: The number of passes required in algorithm MultipleRandomWalk in any given phase is the same as in SingleRandomWalk, no matter how many walks are being run in that phase. So the total passes required for a given phase is $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$ again using Claim 2.3.5. The number of phases run is $\log K$. It follows that the total number of passes in MultipleRandomWalk is $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$. Setting $w = \sqrt{\frac{l}{\alpha}}$ and $s = \sqrt{l\alpha}$, the theorem follows. □

We now show how MultipleRandomWalk algorithm can be used for estimating the mixing time.

2.5 Estimating mixing time

In our previous algorithms, we need to preform walks of length l equal to the mixing time, to estimate the PageRank distribution. We now present an algorithm to estimate the mixing time of a graph. However, instead of computing the exact mixing time, we compute the time required for *approximate* mixing of a random walk. That

is, we compute a length l such that running a random walk for l steps from an initial distribution ends at a node with a probability distribution that is *close* to the stationary distribution. We wish to compute l given an initial distribution u. We denote the mixing time for an initial distribution u to reaching within ϵ in L_1 distance of the steady state distribution by $l_u(\epsilon)$. The following definition makes this precise for undirected graphs.

Definition 2.5.1 (ϵ-near mixing time of undirected graph)**.** *We say that $l_u(\epsilon)$ is the* ϵ-near mixing time *of an undirected graph for an initial distribution u if the L_1-distance between the steady state distribution and the distribution obtained after a random walk of length $l_u(\epsilon)$ starting at node from distribution u is at most ϵ. Further, $l_u(\epsilon)$ must be the shortest such length that satisfies this condition.*

For directed graphs, we have a weaker definition of *ϵ-near mixing time.*

Definition 2.5.2 (ϵ-near mixing time of directed graph)**.** *We say that $l_u(\epsilon)$ is the* ϵ-near mixing time *of a directed graph for an initial distribution u if the L_1-distance between the distribution obtained after a random walk of length $l_u(\epsilon)$ starting at a node from distribution u, and the distribution obtained after a random walk of length $l_u(\epsilon) + poly(1/\epsilon)$, is at most ϵ.*

Lemma 2.5.3. *The ϵ-near mixing time is monotonic property, i.e., if given a source distribution, the distribution after a walk of length l is within ϵ in L_1 distance of the steady state distribution, then so is a walk of length $l + 1$.*

Proof. The monotonicity follows from the fact that $||xA||_1 \leq ||x||_1$ for any transition probability matrix A and for any vector x. This in turn follows from the fact that the sum of entries of any column of A is 1.

Now let π be the stationary distribution of the transition matrix A. This implies that if l is ϵ-near mixing, then $||uA^l - \pi||_1 \leq \epsilon$, by definition of ϵ-near mixing time.

Now consider $||uA^{l+1} - \pi||_1$. This is equal to $||uA^{l+1} - \pi A||_1$ since $\pi A = \pi$. However, this reduces to $||(uA^l - \pi)A||_1 \leq \epsilon$. It follows that $(l+1)$ is ϵ-near mixing. □

In this section, specifically, we show the following result.

Theorem 2.5.4. *Given u, one can find a time that is between $l_u(6\epsilon)$ and $l_u(\frac{\epsilon^3}{4\sqrt{n}\log n})$ in $\tilde{O}(n\alpha + poly(\epsilon^{-1})(\sqrt{\frac{M_u n}{\alpha}} + M_u\alpha\sqrt{n}))$ space and $\tilde{O}(\sqrt{\frac{M_u}{\alpha}})$ passes over the stream, where $M_u = l_u(\frac{\epsilon^3}{4\sqrt{n}\log n})$. For directed graphs, one can find a time that is between $l_u(\epsilon)$ and $l_u(\max\{\epsilon^2/(32n^{1/3}), \epsilon/(4\sqrt{n})\})$, in $\tilde{O}(\sqrt{\frac{M_u}{\alpha}})$ passes $\tilde{O}(n\alpha + poly(\epsilon^{-1})(n^{2/3}\sqrt{\frac{M_u}{\alpha}} + M_u\alpha n^{2/3}))$ space, where $M_u = l_u(\max\{\epsilon^2/(32n^{1/3}), \epsilon/(4\sqrt{n})\})$.*

The naïve approach to compute the mixing time requires $O(n)$ space and $O(M_u)$ passes over the input stream This computes uA^{M_u} exactly where u is the initial vector of size n and A the matrix representation of the graph. It takes n space to maintain this vector, and M_u passes to multiply by A once in every pass.

The main idea in estimating the mixing time is to run many random walks of length l from the specified source using the approach described in the previous section, and use these to compute the distribution after l-length random walk. We then compare the distribution at different l, with the stationary distribution, to check if the two distributions are ϵ-near. We need to address the following issues. First, we do not know what value(s) of l to try. Second, we need to compare these distributions with the steady state distribution; while the steady state distribution is easy to compute for an undirected graph, it is hard to compute for directed graphs.

To compare two distributions, we use the technique of Batu et. al. [21] to determine if the distributions are ϵ-near. Their result is summarized in the following theorem.

Theorem 2.5.5 ([21])**.** *Given $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from a black-box distribution X over $[n]$, and a specified distribution Y, there is a test that outputs PASS with high probability if $|X - Y|_1 \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, and outputs FAIL with constant probability*

if $|X - Y|_1 > 6\epsilon$. *Similarly, given two black-box distributions* X *and* Y *over* $[n]$, *there is a test that requires* $O(n^{2/3}\epsilon^{-4}\log n \log(1/\delta))$ *samples which outputs PASS with probability at least* $1-\delta$ *if* $|X - Y|_1 \leq \max\{\epsilon^2/(32n^{1/3}), \epsilon/(4\sqrt{n})\}$, *and outputs FAIL with probability at least* $1-\delta$ *if* $|X - Y|_1 > \epsilon$.

We are now ready to prove Theorem 2.5.4.

Proof of Theorem 2.5.4. For undirected graphs, the stationary distribution of the random walk is well-known to be $\frac{deg(i)}{2m}$ for node i with degree $deg(i)$, where m is the number of edges in the graph. We only need $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from a distribution to compare it to the stationary distribution. This can be achieved by running MultipleRandomWalk to obtain $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ random walks. To find the approximate mixing time, we try out increasing values of l that are powers of 2. Once we find the right consecutive powers of 2, the monotonicity property admits a binary search to determine the exact value of ϵ-near mixing time. Note that we can apply binary search as ϵ-near mixing time is a monotonic property.

The result in [21] also provides an approach to determine if two unknown distributions X and Y over $[n]$ are ϵ-close in L_1 norm; however, this requires $\tilde{O}(n^{2/3}poly(\epsilon^{-1}))$ samples from each distribution. This completes the proof. □

Theorem 2.5.4 gives some interesting consequences for specific values of α and M_u. We state some below. In the extreme cases of $\alpha = 1$ and $\alpha = \frac{1}{M_u}$, we can calculate the ϵ-near mixing time with either of the trade-offs presented in the following corollary.

Corollary 2.5.6. *Given* u, *one can find a time that is between* $l_u(6\epsilon)$ *and* $l_u(\frac{\epsilon^3}{4\sqrt{n}\log n})$ *in either* $\tilde{O}(n)+poly(\epsilon^{-1})(M_u\sqrt{n})$ *space and* $\tilde{O}(\sqrt{M_u})$ *passes, or* $\tilde{O}(\frac{n}{M_u}+M_u\sqrt{n}poly(\epsilon^{-1}))$ *space and* $\tilde{O}(M_u)$ *passes, where* $M_u = l_u(\max\{\epsilon^2/(32n^{1/3}), \epsilon/(4\sqrt{n})\})$

Assuming the actual mixing time of the graph (worst case over all source nodes), say M, is within constant factors of the estimated mixing time, one can compute a

square-root approximation to the conductance Φ of the graph as $\Theta(1/M) \leq \Phi \leq \Theta(1/\sqrt{M})$ as shown in [90]. The conductance of a graph G is defined as $\phi(G) = \min_{S:|S|\leq|V|/2} \frac{E(S,V(G)\setminus S)}{E(S)}$ where $E(S, V(G) \setminus S)$ is the number of the edges spanning the cut, and $E(S)$ is the number of edges on the smaller side of the cut.

2.6 Estimating distributions with better accuracy

In this section, we present an algorithm that has a better space complexity when the accuracy parameter ϵ is less than $\sqrt{l}/n$. The main idea is to replace the estimation of r_i's in MultipleRandomWalk, that measures the probability that node i is used as a center, by another quantity q_i that measures how many w length walks from node i may be used. We start with a modification of SingleRandomWalk, where we assume that there are infinitely many w length walks out of each sampled node, that can be accessed as an oracle. We then look at the expected number (q_i) of w length walks that will be used for each node i. Subsequently, in ModifiedMultipleRandomWalk, we estimate q_i in phases by doubling the number of walks in each phase getting better accuracy each time (just as in MultipleRandomWalk).

We begin by describing the modification to the algorithm SingleRandomWalk. The main difference is that there are many w length walks available from the sampled nodes; so the algorithm gets stuck only when it hits a non-sampled node. This is different from the earlier version where the algorithm got stuck even if it visited a sampled node for the second time. The notation in the algorithm below uses T to denote the sampled nodes obtained by sampling each node with probability α independently. The table W stores infinitely many w length walks per node in T; $W[t, k]$ is the end point of the k'th w length walk starting at t. At most $count[t]$ of these walks is actually used. Note that this table can be populated in l passes, however the space requirement depends on the maximum $count[t]$ for every t. Right now we assume that the table W is infinite and we can obtain a w length walk for

Algorithm 4 ModifiedSingleRandomWalk(u, l)

1: **Input:** Starting node u, and desired walk length l. Array $W[t, k]$ for $t \in T$, $0 < k \leq \infty$ of infinitely many random walks of length w for every node in T; $W[t, k]$ denotes the $k^t h$ walk for node t.
2: **Output:** $\mathcal{L}_u$ the random walk from u of length l.
3: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
4: Let $count[t]$ denote the next unused w length walk of t we will use. Initialize $count[t] = 1$ for all t.
5: Initialize $\mathcal{L}_u$ to a zero length walk starting at u. Let $x \leftarrow u$, the source node.
6: **while** $|\mathcal{L}_u| < l$ **do**
7:
1. if $(x \in T)$ extend $\mathcal{L}_u$ by appending the walk corresponding to $W[x, count[x]]$. $count[x] \leftarrow count[x] + 1$. $x \leftarrow$ new end point of $\mathcal{L}_u$. {we increment $count[x]$ so that the next time the walk ends at x, we will use the next w length walk from x stored in the table W.}

2. if $(x \notin T)$ ModifiedHandleStuckNode(x, T, $\mathcal{L}_u$, l). {this means that x was not in the initial set of sampled nodes}

8: **end while**

any value of $count[t]$. Unlike in SingleRandomWalk where we defined the set S to keep track of all nodes in T whose w length walks get used up, in ModifiedSingleRandomWalk, we do not need S. The algorithm continues extending the walk using the w length walks implicitly stored in the table W until it finds a stuck node. The module ModifiedHandleStuckNode proceeds by sampling s edges out of R where R is the set of stuck nodes visited in the current invocation. In this case, only a non-sampled node can be a stuck node.

With the algorithm in place, we now need to define q_i, that we use instead of p_i, in ModifiedMultipleRandomWalk. Assume a given set of sampled nodes T.

Definition 2.6.1 (q_i)**.** *For any sampled node i, define q_i to be the expected value of $count[i]$ at the end of the execution of l-length random walk using* ModifiedSingleRandomWalk*.*

Notice that the main difference in ModifiedSingleRandomWalk as compared to SingleRandomWalk is that there is no set S to store the *centers* whose w length

Algorithm 5 ModifiedHandleStuckNode(x, T, $\mathcal{L}_u$, l)

1: $R \leftarrow x$.
2: **while** $|\mathcal{L}_u| < l$ **do**
3: $E \leftarrow$ sample s edges (with repetition) out of each node in R.
4: Extend $\mathcal{L}_u$ as far as possible by walking along the sampled edges in E (on visiting a node in R for the k-th time, use the k-th edge of the s sampled edges from that node).
5: $x \leftarrow$ new end point of $\mathcal{L}_u$ after the extension. One of the following cases arise.
   1. if ($x \in R$) **continue** {no new node is seen}
   2. if ($x \in T$) **return** {this means that x is a sampled node; therefore, we can use the next w-length walk from x by accessing the table W}
   3. if ($x \notin T$ and $x \notin R$) $R \leftarrow R \cup \{x\}$. {this means that x is a new node that has not been visited in this invocation, and x is not in the initial set sampled nodes T}
6: **end while**

walk has been used up. Every node in the initial sampled set T has sufficient number of w length walks. So whenever this walk gets stuck it is stuck at a *non-sampled node*. In this case, s edges are sampled out of all the new nodes visited since the last time a w length walk was used; as before this set is denoted by R.

Estimating q_i *using* SingleRandomWalk - We now show how to estimate q_i. In the first phase, we use SingleRandomWalk to perform $O(\log n)$ walks. To estimate the q_i's, the entire walks of length l need to be reconstructed. This can be done using the pseudo-random coin tosses as described in Remark 2.3.7 which adds $\tilde{O}(l)$ to the space requirements. Once the walks of length l are reconstructed, for any walk, start walking from the source node; each time a sampled node (say t) is seen, increment $count[t]$ and skip w steps, and continue walking till the end. Set the estimate $\tilde{q_i}$ to be the average of $count[i]$ over the $O(\log n)$ random walks. In phase $j+1$, we run $K_{j+1} = 2^j O(\log n)$ walks and use this to obtain improved estimates of q_i for the next phase.

By definition of q_i, $\sum q_i \leq \frac{l}{w}$. Since $\frac{q_i}{l/w}$ is a random variable in the range $[0,1]$, in K_{j+1} walks the estimate $\frac{\tilde{q}}{l/w}$ is within an additive error of $\sqrt{(\frac{q_i}{2^j l/w})} + \frac{1}{2^j}$ w.h.p.; So

Algorithm 6 ModifiedMultipleRandomWalk($\mathcal{I}$, l, K)

1: **Input**: Distribution of the source nodes, $\mathcal{I}$, and length of the walk, l
2: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
3: Perform phases 1 through $\log K$ as follows -
4: **Phase 1**: Perform $O(\log n)$ walks of length l using SingleRandomWalk. Estimate q_i using the technique described above for reconstructing the walks and tracking $count[i]$ for all $i \in T$. Set estimate $\tilde{q}_i$ for q_i to be the average of $count[i]$ over all the $\log n$ walks.
5: **Phase** $(j+1)$**:** {spawn K_{j+1} walks}
- In w passes, sample $K_{j+1} = \tilde{O}(2^j q_i + \frac{l}{w})$ walks from all $i \in T$.
- Perform $2^j O(\log n)$ random walks using ModifiedSingleRandomWalk and again compute the estimates $\tilde{q}_i$'s. The estimate $\tilde{q}_i$ is obtained by taking the average value of $count[i]$ over the K_{j+1} executions.

after phase $j+1$, our estimate $\tilde{q}_i$ is within $q_i \pm \sqrt{(\frac{q_i}{2^j})\frac{l}{w}} \pm \frac{1}{2^j}\frac{l}{w}$ w.h.p.

Theorem 2.6.2. ModifiedMultipleRandomWalk *can be used to perform K random walks of length l in $\tilde{O}(\sqrt{l/\alpha})$ passes and $\tilde{O}(\alpha n\sqrt{l\alpha} + K\sqrt{\frac{l}{\alpha}} + l)$ space.*

Proof. **Correctness:** The proof is similar to to that of theorem 2.4.1. We need to show that the number of w length walks we store from the sampled nodes in step 1 of each phase $j+1$ in MultipleRandomWalk is *sufficient* for K_{j+1} executions of SingleRandomWalk. Note that (assuming previous phases have completed successfully) in each phase the estimate $\frac{\tilde{q}_i}{l/w}$ is accurate up to an additive error of $\sqrt{(\frac{q_i}{2^j})\frac{l}{w}} \pm \frac{1}{2^j}\frac{l}{w}$ w.h.p. Simplifying as in the proof of theorem 2.4.1, we get that $\frac{q_i}{l/w}$ is at most $O(\frac{\tilde{q}_i}{l/w} + 1/2^j)$ or $q_i = O(\tilde{q}_i + \frac{l}{w2^j})$ w.h.p. The number of walks in phase $j+1$ that use this node as a center is w.h.p., by Chernoff bounds, at most $O(K_{j+1}q_i + \log n) \leq O(2^{j+1}\tilde{q}_i \log n + 2l/w + \log n) = \tilde{O}(2^j r_i + l/w)$. This is exactly the number of w-length walks we sample in phase $j+1$. This implies that w.h.p. any time one of $K_{j+1} = 2^j O(\log n)$ walks hits a sampled node, there is an unused w length walk for that node, to extend it. No w length walk or sampled edge, is ever reused throughout the execution; therefore, all random choices in the eventual walk

are independent.

Space: Suppose we do t phases such that $2^t = K$. Total space required would include $O(n\alpha)$ space to store the sampled nodes; to store the w length walks in the phase $(t-1)$, we need $\tilde{O}(\sum_{i\in T}(2^t q_i + \frac{l}{w})) = \tilde{O}(K\sum_{i\in T} q_i + n\alpha\frac{l}{w}) = \tilde{O}(K(\frac{l}{w}) + n\alpha\frac{l}{w})$ space; and finally, the space for sampling s edges in each execution of ModifiedHandleStuckNode amounts to $\tilde{O}(K\frac{1}{\alpha}s)$. Observe that in ModifiedHandleStuckNode, we only sample s edges out of R instead of $S \cup R$ as in HandleStuckNode. Also, as shown before, $|R| \leq \tilde{O}(1/\alpha)$ since each new node is likely to be a sampled node with probability α. So, the space required for the executions of ModifiedhandleStuckNode is $\tilde{O}(K\frac{1}{\alpha}s)$. Also the first phase required $\tilde{O}(l)$ space to reconstruct the walks obtained from SingleRandomWalk. Therefore, the total space required for this algorithm is $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s + \alpha n\frac{l}{w} + l)$.

Passes: The number of passes required is same as in MultpleRandomWalk. So the total passes required for this walk is still $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$.

Setting $s = \sqrt{l\alpha}$ and $w = \sqrt{l}\alpha$ completes the proof. □

Comparing this with Theorem 2.4.1, we see that in the space requirement, the term $Kl\alpha$ is no longer there; however, the space of $O(n\alpha)$ in Theorem 2.4.1 has increased to $O(\alpha n\sqrt{l\alpha})$; additionally, for the first phase where we needed to reconstruct the walks, we incurred an additional space requirement of $O(l)$. Depending on the values and bounds required, one of these theorems is a better than the other. This gives the following corollary similar to Corollary 2.4.3.

Corollary 2.6.3. *For any node with probability p in the probability distribution after a walk of length l, one can approximate its probability up to an accuracy of $p \pm \sqrt{p\epsilon} \pm \epsilon$ for any $\epsilon > 0$ in $\tilde{O}(\sqrt{l/\alpha})$ passes and $\tilde{O}(\alpha n\sqrt{l\alpha} + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + l)$ space. This also implies a $(1 \pm \sqrt{\frac{\epsilon}{p}} \pm \frac{\epsilon}{p})$-approximation ratio for any node with value p in the probability distribution.*

Notice that this is a constant (close to 1) factor approximation for any node with $p \gg \epsilon$.

2.7 Conclusions

We presented the following results for graphs presented as edge streams:

1. Algorithm SingleRandomWalk to perform a random walk of length l in $O(\sqrt{l/\alpha})$ passes and $O(n\alpha + \sqrt{l/\alpha})$ space.

2. Algorithm MultipleRandomWalk and algorithm ModifiedMultipleRandomWalk can perform K random walks of length l in $\tilde{O}(\sqrt{l/\alpha})$ passes and $\tilde{O}(n\alpha + K\sqrt{l/\alpha} + Kl\alpha)$ space or $\tilde{O}(n\alpha\sqrt{l\alpha} + K(\sqrt{l/\alpha}))$ space respectively. These algorithms also provide an approach to approximating the probability distribution after a random walk of length l. It follows that every node with probability p in the probability distribution after a random walk of length l can be approximated to an error of $\pm\sqrt{p\epsilon} \pm \epsilon$ using $\tilde{O}(\sqrt{l/\alpha})$ passes and $\min\{\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{l/\alpha} + \frac{1}{\epsilon}l\alpha), \tilde{O}(n\alpha\sqrt{l\alpha} + (1/\epsilon)(\sqrt{l/\alpha}))\}$ space. In particular, the latter algorithm performs better for thresholds $\epsilon \leq \sqrt{l}/n$.

3. We use this technique and present an approach to determine the ϵ-near mixing time.

Some open questions that arise are:

1. Can we estimate the distribution of nodes with accuracy $\epsilon = 1/n$ using $O(n)$ space?

2. Can one prove any space-pass trade-off bounds? The trivial algorithm to calculate the exact distribution after a random walk of length l requires $O(nl)$ in the space×passes product. Our result stated in Corollary 2.6.3, for a threshold of $\epsilon = 1/n$, also has the same space-pass trade-off.

3. Is there a lower bound on the number of passes required to perform random walks or estimate distributions when the space allowed is $O(n)$? Alternatively, is there an $O(n)$ space constant pass algorithm?

CHAPTER III

SUBLINEAR ROUND ALGORITHM FOR RANDOM WALKS IN DISTRIBUTED NETWORKS

Performing random walks in networks is a fundamental primitive that has found applications in many areas of computer science, including distributed computing. In this chapter, work in [51], we focus on the problem of performing random walks efficiently in a distributed network. Given bandwidth constraints, the goal is to minimize the number of rounds required to obtain a random walk sample.

All previous algorithms that compute a random walk sample of length ℓ as a subroutine always do so naively, i.e., in $O(\ell)$ rounds. The main contribution of this work [51] is a fast distributed algorithm for performing random walks. We show that a random walk sample of length ℓ can be computed in $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds on an undirected unweighted network, where D is the diameter of the network.[1] When $\ell = \Omega(D \log n)$, this is an improvement over the naive $O(\ell)$ bound. (We show that $\Omega(\min\{D, \ell\})$ is a lower bound and hence in general we cannot have a running time faster than the diameter of the graph.) We also show that our algorithm can be applied to speedup the more general Metropolis-Hastings sampling.

We extend our algorithms to perform a large number, k, of random walks efficiently. We show how k destinations can be sampled in $\tilde{O}((k\ell)^{2/3}D^{1/3})$ rounds if $k \leq \ell^2$ and $\tilde{O}((k\ell)^{1/2})$ rounds otherwise. We also present faster algorithms for performing random walks of length larger than (or equal to) the mixing time of the underlying graph. Our techniques can be useful in speeding up distributed algorithms for a variety of applications that use random walks as a subroutine.

[1] $\tilde{O}$ hides $\frac{logn}{\delta}$ factors where n is the number of nodes in the network and δ is the minimum degree.

3.1 Related Work and Contributions

Random walks play a central role in computer science, spanning a wide range of areas in both theory and practice, including distributed computing. Algorithms in many different applications use random walks as an integral subroutine. Applications in networks include token management [89, 23, 44], load balancing [93], small-world routing [101], search [143, 2, 42, 72, 114], information propagation and gathering [24, 97], network topology construction [72, 105, 111], checking expander [56], constructing random spanning trees [32, 17, 16], monitoring overlays [124], group communication in ad-hoc network [57], gathering and dissemination of information over a network [6], distributed construction of expander networks [105], and peer-to-peer membership management [68, 144]. Random walks have also been used to provide uniform and efficient solutions to distributed control of dynamic networks [35]. The paper of [143] describes a broad range of network applications that can benefit from random walks in dynamic and decentralized settings. For further references on applications of random walks to distributed computing, see, e.g. [35, 143].

A key purpose of random walks in many of these network applications is to perform node sampling. Random walk-based sampling is simple, local, and robust. While the sampling requirements in different applications vary, whenever a true sample is required from a random walk of certain steps, all applications perform the walks naively. In this work we present the first non-trivial distributed random walk sampling algorithms in arbitrary networks that are significantly faster than the existing (naive) approaches.

3.1.1 Problems

Although using random walks help in improving the performance of many distributed algorithms, all known algorithms perform random walks naively: Each walk of length ℓ is performed by sending a token for ℓ steps, picking a random neighbor with each

step. Is there a faster way to perform a random walk distributively? In particular, we consider the following *basic* random walk problem.

Computing One Random Walk where Destination Outputs Source. Let s be any node in the network. We want a distributed algorithm such that, in the end, one node v outputs the ID of s where v is randomly picked according to the probability that it is the destination of a random walk of length ℓ starting at s (the source node). We want an algorithm that finishes in the smallest number of rounds.

We consider the following generalizations to the problem.

1. *k Random Walks, Destinations output Sources (k-RW-DoS)*: We have k sources $s_1, s_2, ..., s_k$ (not necessarily distinct) and we want each of k destinations to output an ID of its corresponding source.

2. *k Random Walks, Sources output Destinations (k-RW-SoD)*: Same as above but we want each source to output the ID of its corresponding destination.

It turns out that solving k-RW-SoD can be more expensive than solving k-RW-DoS. An extension of the first problem can be used in applications where the sources only want to know a "synopsis" of the destination, such as aggregating statistics and computing a function (max load, average load) by sampling nodes. The second problem is used when sources want to know data of each destination separately.

To demonstrate that these problems are non-trivial, let us first focus on the basic random walk problem (which is equivalent to 1-RW-DoS). The following naive algorithm finishes in $O(\ell)$ rounds: Circulate a token (with ID of s written on it) starting from s for ℓ rounds (in each round, the node having the token currently, forwards it to a random neighbor) and, in the end, the vertex v that holds the token outputs the ID of s. Our goal is to devise algorithms that are faster than this ℓ-round algorithm. To achieve faster algorithms, a node cannot just wait until it receives the token and forwards it. It is necessary to "forward the token ahead of time". One natural approach

is to guess which nodes will be in the walk and ask them to forward the token ahead of time. However, even if one knew how many times each node is expected to be seen on the walk (without knowing the order), it is still not clear what running time one can guarantee. The difficulty is that many pre-forwarded tokens may cause congestion. A new approach is needed to obtain fast distributed computation of random walks. We present the first such results in this work.

Notation: Throughout the work, we let ℓ be the length of the walks, k be the number of walks, D be the network diameter, δ be the minimum node degree and n be the number of nodes in the network.

3.1.2 Distributed Computing Model

Before we present our results, we describe our model which is standard in the literature. Without loss of generality, we assume that the graph is connected. Each node has a unique identifier and at the beginning of the computation, each node v accepts as input its own identifier. The nodes are allowed to communicate (only) through the edges of the graph G. We assume that the communication is synchronous and occurs in discrete rounds (time steps). We assume the $\mathcal{CONGEST}$ communication model, a widely used standard model to study distributed algorithms [131, 128]: a node v can send an arbitrary message of size at most $O(\log n)$ through an edge per time step. (We note that if unbounded-size messages were allowed through every edge in each time step, then the problems addressed here can be trivially solved in $O(D)$ time by collecting all the topological information at one node, solving the problem locally, and then broadcasting the results back to all the nodes [131].) It is typically straightforward to generalize the results to a $\mathcal{CONGEST}(B)$ model, where $O(B)$ bits can be transmitted in a single time step across an edge. Our time bounds (measured in number of rounds) are for the synchronous communication model. However, our algorithms will also work in an asynchronous model under the same asymptotic time

bounds, using a standard tool called the *synchronizer* [131]. We assume that all nodes start simultaneously.

3.1.3 Main Contributions

We present the first non-trivial distributed algorithms for computing random walks (both k-RW-DoS and k-RW-SoD) in undirected, unweighted graphs. First, for 1-RW-DoS (cf. Section 4.2), we give an $O(\frac{\ell^{2/3}D^{1/3}(\log n)^{1/3}}{\delta^{1/3}})$-round algorithm. Many real-world networks (e.g., peer-to-peer networks) have small diameter D and random walks of length at least the diameter are usually performed; that is, $l \gg D$. In this case, the algorithm above finishes in roughly $\tilde{O}(\ell^{2/3})$ rounds, which is a significant improvement over the naive $O(\ell)$ round algorithm. The main idea behind our $O(\frac{\ell^{2/3}D^{1/3}(\log n)^{1/3}}{\delta^{1/3}})$-round algorithm is to "prepare" a few short walks in the beginning and carefully stitch these walks together later as necessary. If there are not enough short walks, we construct more of them on the fly. We overcome a key technical problem by showing how one can perform many short walks in parallel without causing too much congestion. Our results also apply to the cost-sensitive model [15] on weighted graphs.

We then present extensions of our algorithm to perform random walk according to the Metropolis-Hastings [85, 122] algorithm, a more general type of random walk with numerous applications (e.g., [143]). The Metropolis-Hastings algorithm gives a way to define transition probabilities so that a random walk converges to any desired distribution. For an important special case, when the desired distribution is uniform, our time bounds reduce to the same as above.

The above algorithms can be extended to solve k-RW-DoS (cf. Section 3.3) in $O(k \cdot \frac{\ell^{2/3}D^{1/3}(\log n)^{1/3}}{\delta^{1/3}})$ rounds straightforwardly. However, we show that, with a small modification, one can do much better. We give algorithms for two cases. When the goal is to perform a few walks, i.e. $k \leq \ell^2$, we can do this in $O(\frac{(k\ell)^{2/3}D^{1/3}(\log n)^{1/3}}{\delta^{1/3}})$

rounds. Moreover, when $k \geq \ell^2$, one can do this in only $O(\sqrt{\frac{k\ell \log n}{\delta}})$ rounds. We then give a simple algorithm for an important special case, namely when $\ell \geq t_{mix}$ where t_{mix} is the mixing time of the underlying graph (cf. Section 3.5). We develop an $O(D+k)$ algorithm for k-RW-DoS and k-RW-SoD. We also observe that $\Omega(\min(\ell, D))$ is a straightforward lower bound. Therefore, we have tight algorithms when $\ell \leq D$ or $\ell \geq t_{mix}$ and, for $D \leq \ell \leq t_{mix}$, we have efficient non-trivial algorithms.

Finally, we extend the k-RW-DoS algorithms to solve the k-RW-SoD problem (cf . Section 3.4) in additional $O(k)$ rounds. We also observe that $\Omega(k + \min(\ell, D))$ is a lower bound on the number of rounds required. Therefore, our algorithms for k-RW-SoD are asymptotically tight. We note an interesting fact that algorithms for k-RW-DoS finishes in $o(k)$ rounds when k is much larger than ℓ and D while k is the lower bound of k-RW-SoD.

We note that the focus of this work is on the time complexity and not on the message (communication) complexity of performing random walks. In general, the message complexity of our algorithms can be larger than the message complexity of the naive random walk algorithm (that takes only ℓ messages to perform a walk of length ℓ).

3.1.4 Applications and Related Work

Random walks have been used in a wide variety of applications in distributed networks as mentioned in the beginning. We describe here some of the applications in more detail.

Speeding up distributed algorithms using random walks has been considered for a long time. Besides our approach of speeding up the random walk itself, one popular approach is to reduce the *cover time*. Recently, Alon et. al. [9] show that performing several random walks in parallel reduces the cover time in various types of graphs. They assert that the problem with performing random walks is often the latency. In

these scenarios where many walks are performed, our results could help avoid too much latency and yield an additional speed-up factor.

A nice application of random walks is in the design and analysis of expanders. We mention two results here. Law and Siu [105] consider the problem of constructing expander graphs in a distributed fashion. One of the key subroutines in their algorithm is to perform several random walks from specified source nodes. While the overall running time of their algorithm depends on other factors, the specific step of computing random walk samples can be improved using our techniques presented in this work. Dolev and Tzachar [56] use random walks to check if a given graph is an expander. The first algorithm given in [56] is essentially to run a random walk of length $n \log n$ and mark every visited vertices. Later, it is checked if every node is visited. It can be seen that our algorithm implies that the first step can be done in $\tilde{O}((n \log n)^{2/3} D^{1/3})$ rounds.

Broder [32] and Wilson [140] gave algorithms to generate random spanning trees using random walks and Broder's algorithm was later applied to the network setting by Bar-Ilan and Zernik [17]. Recently Goyal et al. [77] show how to construct an expander/sparsifier using random spanning trees. If their algorithm is implemented on a distributed network, the techniques presented in this work would yield an additional speed-up in the random walk constructions.

Morales and Gupta [124] discuss about discovering a consistent and available monitoring overlay for a distributed system. For each node, one needs to select and discover a list of nodes that would monitor it. The monitoring set of nodes need to satisfy some structural properties such as consistency, verifiability, load balancing, and randomness, among others. This is where random walks come in. Random walks is a natural way to discover a set of random nodes that are spread out (and hence scalable), that can in turn be used to monitor their local neighborhoods. Random walks have been used for this purpose in another paper by Ganesh et al. [68] on

peer-to-peer membership management for gossip-based protocols.

The only work that uses the same general approach as this work is the work presented in the previous Chapter 2 and the corresponding paper is [48]. They consider the problem of finding random walks in data streams with the main motivation of finding PageRank. The same general idea of stitching together short walks is used. They consider the model where the graph is too big to store in main memory, and the algorithm has *streaming* access to the edges of the graph while maintaining limited storage. They show how to perform ℓ length random walks in about $\sqrt{\ell}$ passes over the data. This improves upon the naive ℓ pass approach and thereby leads to improved algorithms for estimating PageRank vectors. The distributed setting considered in this work has very different constraints and motivations from the streaming setting and calls for new techniques. Recently, Sami and Twigg [133] consider lower bounds on the communication complexity of computing stationary distribution of random walks in a network. Although, their problem is related to our problem, the lower bounds obtained do not imply anything in our setting.

3.2 Algorithm for one random walk

3.2.1 Description of the Algorithm

In this section, we present the main ideas of our approach by developing an algorithm for 1-RW-DoS called Single-Random-Walk (cf. Algorithm 13) for undirected graphs. The naive upper and lower bounds for 1-RW-DoS are ℓ and D respectively (the lower bound is formalized later in this work). We present the first nontrivial upper bound, i.e., perform walks of length $\ell > D$ in fewer than ℓ rounds. Single-Random-Walk runs with two important variables: η and λ. The main idea is to first perform η random walks of length λ from every node. Subsequently, starting at the source node s, these λ length walks are "stitched" to form a longer walk (traversing a new walk of length λ from the end point of the previous walk of

Algorithm 7 SINGLE-RANDOM-WALK(s, ℓ)

Input: Starting node s, and desired walk length ℓ.
Output: Destination node of the walk outputs the ID of s.

Phase 1: (Each node performs η random walks of length λ)

1: Each node constructs η (identical) messages containing its ID and a counter which is initialized to 0.
2: **for** $i = 1$ to λ **do**
3: This is the i-th iteration. Each node v does the following: Consider each message M held by v and received in the $(i-1)$-th iteration (having current counter $i-1$). Pick a neighbor u uniformly at random and forward M to u after incrementing its counter. {Note that any iteration could require more than 1 round.}
4: **end for**

Phase 2: (Stitch ℓ/λ walks of length λ)

1: s creates a message called "token" with the ID of s
2: **for** $i = 1$ to $\lfloor \ell/\lambda \rfloor$ **do**
3: Let v be a node that is currently holding a token
4: v calls SAMPLE-DESTINATION(v) and let v' be the returned value (which is a destination of an unused random walk of length λ starting at v)
5: **if** $v' =$ NULL (all walks from v have already been used up) **then**
6: v calls GET-MORE-WALKS(v, η, λ) (Perform η walks of length λ starting at v)
7: v calls SAMPLE-DESTINATION(v) and let v' be the returned value
8: **end if**
9: v sends the token to v'
10: **end for**
11: Walk naively until ℓ steps are completed (this is at most another λ steps).
12: A node holding the token outputs the ID of s

length λ). Whenever a node is visited as an end point of such a walk of length λ, one of its (at most η) *unused* walks is sampled uniformly to preserve randomness. If all η walks from a node have been used up, additional rounds are invested to obtain η more walks from this node. This approach turns out to be round-efficient for three reasons. First, performing the initial set of η walks of length λ from all nodes simultaneously can be done efficiently. Second, we give a technique to perform η walks of length λ from a single node efficiently. Finally, stitching two λ length walks can be done in about D rounds.

Algorithm 8 GET-MORE-WALKS(v, η, λ)

(Starting from node v, perform η number of random walks, each of length λ.)

1: The node v constructs η (identical) messages containing its ID.
2: **for** $i = 1$ to λ **do**
3: Each node u does the following:
4: - For each message M held by u, pick a neighbor z uniformly at random as a receiver of M.
5: - For each neighbor z of u, send ID of v and the number of messages that z is picked as a receiver, denoted by $c(u, v)$.
6: - For each neighbor z of u, up on receiving ID of v and $c(u, v)$, constructs $c(u, v)$ messages, each contains the ID of v.
7: **end for**

We now explain algorithm SINGLE-RANDOM-WALK (cf. Algorithm 13) in some more detail. The algorithm consists of two phases. In the first phase, each node performs η random walks of length λ each. To do this, each node initially constructs η messages with its ID. Then, each node forwards each message to a random neighbor. This is done for λ steps. At the end of this phase, if node u has k messages with the ID of node v, then u is a destination of k walks starting at v. Note that v has no knowledge of the destinations of its own walks. The main technical issue to deal with here is that performing many simultaneous random walks can cause too much congestion. We show a key lemma (Lemma 3.2.7) that bounds the time needed for this phase.

In the second phase, we perform a random walk starting from source s by "stitching" walks of length λ obtained in the first phase into a longer walk. The process goes as follows. Imagine that there is a token initially held by s. Among η walks starting at s (obtained in phase 1), randomly select one. Note that this step is not straightforward since s has no knowledge of the destinations of its walks. Further, selecting an arbitrary destination would violate randomness. (A minor technical point: one may try to use the i-th walk when it is reached for the i-th time; however, this is not possible because one cannot mark tokens separately in GET-MORE-WALKS

(described later), since we only send counts forward to avoid congestion on edges). Sample-Destination algorithm (cf. Algorithm 15) is used to perform this step. We prove in Lemma 4.2.3 that this can be done in $O(D)$ rounds.

When Sample-Destination(v) is called by any node v, the algorithm randomly picks a message v's ID written on it, returns the ID of the node that holds this message, and then deletes it. If there is no such message (e.g., when Sample-Destination(v) has been called η times), it returns null.

Let v receive u_d as an output from Sample-Destination. Notice that v receives the ID of u_d through the edges on the graph, thereby requiring D rounds. v sends the token to u_d and the process repeats. That is, u_d randomly selects a random walk starting at u_d and forwards the token to the destination. If the process continues without Sample-Destination returning null, then a walk of length ℓ will complete after ℓ/λ repetitions.

However, if null is returned by Sample-Destination for v, then the token cannot be forwarded further. At this stage, η more walks of length λ are performed from v by calling Get-More-Walks(v, η, λ) (cf. Algorithm 14). This algorithm creates η messages with ID v and forwards them for λ random steps. This is done fast by only sending counts along edges that require multiple messages. This is crucial in avoiding congestion. While one cannot directly bound the number of times any particular node v invokes Get-more-Walks, a simple amortization argument is used to bound the running time of invocations over all nodes.

3.2.2 Analysis

Theorem 3.2.1. *Algorithm* Single-Random-Walk *(cf. Algorithm 13) solves* 1*-RW-DoS and, with high probability*[2]*, finishes in* $O(\frac{\ell^{2/3}D^{1/3}(\log n)^{1/3}}{\delta^{1/3}})$ *rounds.*

[2] *With high probability means with probability at least* $(1 - \frac{1}{n})$ *throughout this chapter.*

We begin by analyzing the time needed by Phase 1 of Algorithm Single-Random-Walk.

Lemma 3.2.2. *Phase 1 finishes in $O(\frac{\lambda\eta\log n}{\delta})$ rounds with high probability, where δ is the minimum node degree.*

Proof. Consider the case when each node v creates $\eta \cdot \frac{degree(v)}{\delta} \geq \eta$ messages. We show that the lemma holds even in this case. For each message M, any $j = 1, 2, ..., \lambda$, and any edge e, we define $X^j_M(e)$ to be a random variable having value 1 if M is sent through e in the j^{th} iteration (i.e., when the counter on M has value $j-1$). Let $X^j(e) = \sum_{M:\text{message}} X^j_M(e)$. We compute the expected number of messages that go through an edge, see claim below.

Claim 3.2.3. *For any edge e and any j, $\mathbb{E}[X^j(e)] = 2\frac{\eta}{\delta}$.*

Proof. Assume that each node v starts with $\eta \cdot \frac{degree(v)}{\delta} \geq \eta$ messages. Each message takes a random walk. We prove that after any given number of steps j, the expected number of messages at node v is still $\eta\frac{degree(v)}{\delta} \geq \eta$. Consider the random walk's probability transition matrix, call it A. In this case $Au = u$ for the vector u having value $\frac{degree(v)}{2m}$ where m is the number of edges in the graph (since this u is the stationary distribution of an undirected unweighted graph). Now the number of messages we started with at any node i is proportional to its stationary distribution, therefore, in expectation, the number of messages at any node remains the same.

To calculate $\mathbb{E}[X^j(e)]$, notice that edge e will receive messages from its two end points, say x and y. The number of messages it receives from node x in expectation is exactly the number of messages at x divided by $degree(x)$. The lemma follows. □

By Chernoff's bound (e.g., in [123, Theorem 4.4.]), for any edge e and any j,

$$\mathbb{P}[X^j(e) \geq 4\log n\frac{\eta}{\delta}] \leq 2^{-4\log n} = n^{-4}.$$

It follows that the probability that there exists an edge e and an integer $1 \leq j \leq \lambda$ such that $X^j(e) \geq 4 \log n \frac{\eta}{\delta}$ is at most $|E(G)|\lambda n^{-4} \leq \frac{1}{n}$ since $|E(G)| \leq n^2$ and $\lambda \leq \ell \leq n$ (by the way we define λ).

Now suppose that $X^j(e) \leq 4 \log n \frac{\eta}{\delta}$ for every edge e and every integer $j \leq \lambda$. This implies that we can extend all walks of length i to length $i+1$ in $4 \log n \frac{\eta}{\delta}$ rounds. Therefore, we obtain walks of length λ in $4\lambda \frac{\eta}{\delta} \log n$ rounds as claimed. (Note that if $\eta \leq \delta$, we still get a high probability bound for $X^j(e) \geq 4 \log n$.) □ □

We next show the time needed for Get-More-Walks and Sample-Destination.

Lemma 3.2.4. *For any v,* Get-More-Walks(v, η, λ) *always finishes within $O(\lambda)$ rounds.*

Proof. Consider any node v during the execution of the algorithm. If it contains x copies of the source ID, for some x, it has to pick x of its neighbors at random, and pass the source ID to each of these x neighbors. Although it might pass these messages to less than x neighbors, it sends only the source ID and a *count* to each neighbor, where the count represents the number of copies of source ID it wishes to send to such neighbor. Note that there is only one source ID as one node calls Get-More-Walks at a time. Therefore, there is no congestion and thus the algorithm terminates in $O(\lambda)$ rounds. □

Lemma 3.2.5. Sample-Destination *always finishes within $O(D)$ rounds.*

Proof. Constructing a BFS tree clearly takes only $O(D)$ rounds. In the second phase where the algorithm wishes to *sample* one of many tokens (having its ID) spread across the graph. The sampling is done while retracing the BFS tree starting from leaf nodes, eventually reaching the root. The main observation is that when a node receives multiple samples from its children, it only sends one of them to its parent. Therefore, there is no congestion. The total number of rounds required is therefore

the number of levels in the BFS tree, $O(D)$. The third phase of the algorithm can be done by broadcasting (using a BFS tree) which needs $O(D)$ rounds. □

Next we show the correctness of the SAMPLE-DESTINATION algorithm.

Lemma 3.2.6. *Algorithm* SAMPLE-DESTINATION*(v) (cf. Algorithm 15), for any node v, samples a destination of a walk of length λ uniformly at random.*

Proof. Assume that before this algorithm starts, there are t (without loss of generality, let $t > 0$) "tokens" containing ID of v stored in some nodes in the network. The goal is to show that SAMPLE-DESTINATION brings one of these tokens to v with uniform probability. For any node u, let T_u be the subtree rooted at u and let S_u be the set of tokens in T_u. (Therefore, $T_v = T$ and $|S_v| = t$.)

We claim that any node u returns a destination to its parent with uniform probability (i.e., for any tokens $x \in S_u$, $Pr[u \text{ returns } x]$ is $1/|S_u|$ (if $|S_u| > 0$)). We prove this by induction on the height of the tree. This claim clearly holds for the base case where u is a leaf node. Now, for any non-leaf node u, assume that the claim is true for any of its children. Suppose that u receives tokens and counts from q children. Assume that it receives tokens $d_1, d_2, ..., d_q$ and counts $c_1, c_2, ..., c_q$ from nodes $u_1, u_2, ..., u_q$, respectively. (Also recall that d_0 is the sample of its own tokens (if exists) and c_0 is the number of its own tokens.) By induction, d_j is sent from u_j to u with probability $1/|S_{u_j}|$, for any $1 \leq j \leq q$. Moreover, $c_j = |S_{u_j}|$ for any j. Therefore, any token d_j will be picked with probability $\frac{1}{|S_{u_j}|} \times \frac{c_j}{c_0+c_1+...c_q} = \frac{1}{S_u}$.

The lemma follows by applying the claim above to v. □

We are now ready to state and prove the running time of the main algorithm for 1-RW-DoS.

Lemma 3.2.7. *Algorithm* SINGLE-RANDOM-WALK *(cf. Algorithm 13) solves 1-RW-DoS and, with high probability, finishes in $O(\frac{\lambda\eta \log n}{\delta} + \frac{\ell \cdot D}{\lambda} + \frac{\ell}{\eta})$ rounds.*

Proof. First, we prove the correctness of the algorithm. Observe that any two λ-length walks (possibly from different sources) are independent from each other. Moreover, a walk from a particular node is picked uniformly at random (by Lemma 4.6.2). Therefore, the Single-Random-Walk algorithm is equivalent to having a source node perform a walk of length λ and then have the destination do another walk of length λ and so on.

We now prove the time bound. First, observe that algorithm Sample-Destination is called $O(\frac{\ell}{\lambda})$ times and by Lemma 4.2.3, this algorithm takes $O(\frac{\ell \cdot D}{\lambda})$ rounds in total. Next, we claim that Get-More-Walks is called at most $O(\frac{\ell}{\lambda\eta})$ times in total (summing over all nodes). This is because when a node v calls Get-More-Walks(v, η, λ), all η walks starting at v must have been stitched and therefore v contributes $\lambda\eta$ steps of walk to the long walk we are constructing. It follows from Lemma 4.2.2 that Get-More-Walks algorithm takes $O(\frac{\ell}{\eta})$ rounds in total.

Combining the above results with Lemma 4.2.1 gives the claimed bound. □

Theorem 4.2.5 immediately follows.

Proof. Use Lemma 3.2.7 with $\lambda = \frac{\ell^{1/3} D^{2/3} \delta^{1/3}}{(\log n)^{1/3}}$ and $\eta = \frac{\ell^{1/3} \delta^{1/3}}{D^{1/3} (\log n)^{1/3}}$. □

3.2.3 Generalization to the Metropolis-Hastings

We now discuss extensions of our algorithm to perform random walk according to the Metropolis-Hastings algorithm, a more general type of random walk with numerous applications (e.g., [143]). The Metropolis-Hastings [85, 122] algorithm gives a way to define a transition probability so that a random walk converges to any desired distribution π (where π_i, for any node i, is the desired stationary distribution at node i). It is assumed that every node i knows its steady state distribution π_i (and can know its neighbors' steady state distribution in one round). The algorithm is roughly as follows. For any desired distribution π and any desired *laziness factor* $0 < \alpha < 1$, the transition probability from node i to its neighbor j is defined to be

$P_{ij} = \alpha \min(1/d_i, \pi_j/(\pi_i d_j))$ where d_i and d_j are degree of i and j respectively. It is shown that a random walk with this transition probability converges to π. We claim that one can modify the SINGLE-RANDOM-WALK algorithm to compute a random walk with above transition probability. We state the main theorem for this process below.

Theorem 3.2.8. *The* SINGLE-RANDOM-WALK *algorithm with transition probability defined by Metropolis-Hastings algorithm finishes in* $O(\frac{\lambda\eta \log n}{\min_{i,j}(d_i\pi_j/(\alpha\pi_i))} + \frac{\ell D}{\lambda} + \frac{\ell}{\eta})$ *rounds with high probability.*

The proof is similar as in the previous Section. In particular, all lemmas proved earlier hold for this case except Lemma 4.2.1. We state a similar lemma here with proof

Lemma 3.2.9. *For any* π *and* α*, Phase 1 finishes in* $O(\frac{\lambda\eta \log n}{\min_{i,j}(d_i\pi_j/(\alpha\pi_i))})$ *rounds with high probability.*

Proof. The proof is essentially the same as Lemma 4.2.1. We present it here for completeness. Let $\beta = \min_i \frac{d_i}{\alpha\pi_i}$ and let $\rho = \frac{\eta}{\beta \min_i \pi_i}$. Consider the case when each node i creates $\rho\beta\pi_i \geq \eta$ messages. We show that the lemma holds even in this case.

We use the same definition as in Lemma 4.2.1. That is, for each message M, any $j = 1, 2, ..., \lambda$, and any edge e, we define $X_M^j(e)$ to be a random variable having value 1 if M is sent through e in the j^{th} iteration (i.e., when the counter on M has value $j-1$). Let $X^j(e) = \sum_{M:\text{message}} X_M^j(e)$. We compute the expected number of messages that go through an edge. As before, we show the following claim.

Claim 3.2.10. *For any edge* e *and any* j, $\mathbb{E}[X^j(e)] = 2\rho$.

Proof. Assume that each node v starts with $\rho\beta\pi_i \geq \eta$ messages. Each message takes a random walk. We prove that after any given number of steps j, the expected number of messages at node v is still $\rho\beta\pi_i$. Consider the random walk's probability transition

matrix, say A. In this case $Au = u$ for the vector u having value π_v (since this π_v is the stationary distribution). Now the number of messages we started with at any node i is proportional to its stationary distribution, therefore, in expectation, the number of messages at any node remains the same.

To calculate $\mathbb{E}[X^j(e)]$, notice that edge e will receive messages from its two end points, say x and y. The number of messages it receives from node x in expectation is exactly $\rho\beta\pi_x \times \min(\frac{1}{d_x}, \frac{\pi_y}{\pi_x d_y}) \leq \rho$ (since $\beta \leq \frac{d_x}{\alpha\pi_x}$). The lemma follows. □

By Chernoff's bound (e.g., in [123, Theorem 4.4.]), for any edge e and any j,

$$\mathbb{P}[X^j(e) \geq 4\rho \log n] \leq 2^{-4\log n} = n^{-4}.$$

It follows that the probability that there exists an edge e and an integer $1 \leq j \leq \lambda$ such that $X^j(e) \geq 4\rho \log n$ is at most $|E(G)|\lambda n^{-4} \leq \frac{1}{n}$ since $|E(G)| \leq n^2$ and $\lambda \leq \ell \leq n$ (by the way we define λ).

Now suppose that $X^j(e) \leq 4\rho \log n$ for every edge e and every integer $j \leq \lambda$. This implies that we can extend all walks of length i to length $i+1$ in $4\rho \log n$ rounds. Therefore, we obtain walks of length λ in $4\lambda\rho \log n = \frac{\lambda\eta \log n}{\min_{i,j}(d_i\pi_j/(\alpha\pi_i))}$ rounds as claimed. □ □

An interesting application of the above lemma is when π is a uniform distribution. In this case, we can compute a random walk of length ℓ in $O(\frac{\lambda\eta\alpha \log n}{\delta} + \frac{\ell D}{\lambda} + \frac{\ell}{\eta})$ rounds which is exactly the same as Lemma 3.2.7 if we use $\alpha = 1$. In both cases, the minimum degree node causes the most congestion. (In each iteration of Phase 1, it sends the same amount of tokens to each neighbor.)

We end this Section by mentioning two further extensions of our algorithm.

1. **Weighted graphs:** We can generalize our algorithm for weighted graphs, if we assume the cost sensitive communication model of [15]. In this model, we assume that one can send messages proportional to the weights (so weights serve as bandwidth).

This model can be thought of as our model with unweighted multigraph network (note that our algorithm will work for an undirected unweighted multigraph also). The algorithms and analyses are similar.

2. **Obtaining the entire walk:** In the above algorithm, we focus on sampling the destination of the walk and not the more general problem of actually obtaining the entire walk (where every node in the walk knows its position in the walk). However, it is not difficult to extend our approach to the case where each node on the eventual ℓ length walk wants to know its position in the walk within the same time bounds.

3.3 Algorithm for K Random Walks

The previous section is devoted to performing a single random walk of length ℓ efficiently. Many settings usually require a large number of random walk samples. A larger number of samples allows for better estimation of the problem at hand. In this section, we focus on obtaining several random walk samples. For k samples, one could simply run k iterations of the algorithm presented in the previous section; this would require $\tilde{O}(k\ell^{2/3}D^{1/3})$ rounds for k walks. Our algorithms in this section do much better . We consider two different cases, $k \leq \ell^2$ and $k > \ell^2$ and show bounds of $\tilde{O}((k\ell)^{2/3}D^{1/3})$ and $\tilde{O}(\sqrt{kl})$ rounds respectively (saving a factor of $k^{1/3}$). Notice, however, that our results still require greater than k rounds. Our algorithms for the two cases are slightly different; the reason we break this in two parts is because in one case we are able to obtain a stronger result than the other. Before that, we first observe a simple lower bound for the k-RW-DoS problem.

Lemma 3.3.1. *For every $D \leq n$ and every k, there exists a graph G of diameter D such that any distributed algorithm that solves k-RW-DoS on G uses $\Omega(\min(\ell, D))$ rounds with high probability.*

Proof. First, consider when $\ell \geq D$. Let s and t be two nodes of distance exactly D from each other and with only this one path between them. A walk of length

ℓ starting at s has a non-zero probability of ending up at t. In this case, for the source ID of s to reach t, at least D rounds of communication will be required. Using multi-edges, one can force, with high probability, the traversal of the random walk to be along this D length path. Recall that our algorithms can handle multi-edges. This argument holds for 1-RW-DoS. The constructed multigraph can be changed to a simple graph by subdividing each edge with one additional vertex. This way, in expectation, the walk takes two steps of crossing over from one multiedge to another. Now the same argument can be applied for a random walk of length $O(D)$.

Similarly, if $D \geq \ell$ then we consider s and t of distance exactly ℓ apart and apply the same argument. □

3.3.1 k-RW-DoS Algorithm when $k \leq \ell^2$

In this case, the algorithm is essentially repeating algorithm Single-Random-Walk (cf. Algorithm 13) on each source. However, the crucial observation is that we have to do Phase 1 only once.

Theorem 3.3.2. *Algorithm* Few-Random-Walks *(cf. Algorithm 10) solves k-RW-DoS for any $k \leq \ell^2$ and, with high probability, finishes in $O(\frac{(k\ell)^{2/3} D^{1/3} (\log n)^{1/3}}{\delta^{1/3}})$ rounds.*

Proof. The first phase of the algorithm finishes in
$O(\frac{\lambda\eta \log n}{\delta})$ with high probability using Lemma 4.2.1 as it is the same as in 1-RW-DoS. The second phase of Few-Random-Walks takes rounds $O(k(\frac{\ell D}{\lambda}))$. This follows from the argument in Lemma 3.2.7. Essentially, the number of times Sample-Destination will be called by k-RW-DoS is at most k times that by 1-RW-DoS; this is $\frac{k\ell}{\lambda}$. Each call requires $O(D)$ rounds. Finally, Get-More-Walks will be called $\frac{k\ell}{\eta\lambda}$ times, again by the argument in Lemma 3.2.7. Each call requires $O(\lambda)$ rounds, by Lemma 4.2.2. Combining these, the total number of rounds used is $O(\frac{\lambda\eta \log n}{\delta} + k(\frac{\ell D}{\lambda} + \frac{\ell}{\eta}))$.

To optimize this expression, we choose $\eta = \frac{(k\ell\delta)^{1/3}}{(D\log n)^{1/3}}$ and $\lambda = \frac{(k\ell\delta)^{1/3}D^{2/3}}{(\log n)^{1/3}}$, and the result follows. □

We now turn to the second case, where the number of walks required, k, exceeds ℓ^2.

3.3.2 k-RW-DoS Algorithm when $k \geq \ell^2$

When k is large, our choice of λ becomes ℓ and the algorithm can be simplified to be as in algorithm Many-Random-Walks (cf. Algorithm 11). In this algorithm, we do Phase 1 as usual. However, since we use $\lambda = \ell$, we do not have to stitch the short walks together. Instead, we simply check at each source nodes s_i if it has enough walks starting from s_i. If not, we get the rest walks from s_i by calling Get-More-Walk procedure.

Theorem 3.3.3. *Algorithm* Many-Random-Walks *(cf. Algorithm 11) solves k-RW-DoS for any $k \geq \ell^2$ and, with high probability, finishes in $O(\sqrt{\frac{k\ell\log n}{\delta}})$ rounds.*

Proof. We claim that the algorithm finishes in $O(\frac{\eta\ell\log n}{\delta} + k/\eta)$ rounds with high probability. The theorem follows immediately using $\eta = \sqrt{\frac{k\delta}{\ell\log n}}$. Now we prove our claim.

It follows from Lemma 4.2.1 that Phase 1 finishes in $\frac{\eta\ell\log n}{\delta}$ rounds with high probability. Observe that the procedure Get-More-Walk is called at most k/η times and each time it uses at most ℓ rounds by Lemma 4.2.2. □

3.4 k Walks where Sources output Destinations (k-RW-SoD)

In this section we extend our results to k-RW-SoD using the following lemma.

Lemma 3.4.1. *Given an algorithm that solves k-RW-DoS in $O(S)$ rounds, for any S, one can extend the algorithm to solve k-RW-SoD in $O(S + k + D)$ rounds.*

The idea of the above lemma is to construct a BFS tree and have each destination node send its ID to the corresponding source via the root. By using upcast and downcast algorithms [131], this can be done in $O(k + D)$ rounds.

Proof. Let the algorithm that solves k-RW-DoS perform one walk each from source nodes $s_1, s_2, \ldots, s_k$. Let the destinations that output these sources be $d_1, d_2, \ldots, d_k$ respectively. This means that for each $1 \leq i \leq k$, node d_i has the ID of source s_i. To prove the lemma, we need a way for each d_i to communicate its own ID to s_i respectively, in $O(k + D)$ rounds. The simplest way to do this is for each node ID pair (d_i, s_i) to be communicated to some fixed node r, and then for r to communicate this information to the sources s_i. This is done by r constructing a BFS tree rooted at itself. This step takes $O(D)$ rounds. Now, each destination d_i sends its pair (d_i, s_i) up this tree to the root r. This can be done in $O(D + k)$ rounds using an upcast algorithm [131]. Node r then uses the same BFS tree to route back the pairs to the appropriate sources. This again takes $O(D + k)$ rounds using a downcast algorithm [131]. □

Theorem 3.4.2. *Given a set of k sources, one can perform k-RW-SoD after random walks of length ℓ in*
$O(\frac{(k\ell)^{2/3} D^{1/3} (\log n)^{1/3}}{\delta^{1/3}} + D)$ rounds when $k \leq \ell^2$, and in $O(\sqrt{\frac{k\ell \log n}{\delta}} + k + D)$ rounds when $k \geq \ell^2$.

Proof. On applying Lemma 3.4.1 to Theorems 3.3.2 and 3.3.3, we get that k-RW-SoD can be done in $O(\frac{(k\ell)^{2/3} D^{1/3} (\log n)^{1/3}}{\delta^{1/3}} + k + D)$ rounds when $k \leq \ell^2$ and $O(\sqrt{\frac{k\ell \log n}{\delta}} + k + D)$ rounds when $k \geq \ell^2$. Note that when $k \leq \ell^2$, $(k\ell)^{2/3} \geq k$. □

We show an almost matching lower bound below.

Lemma 3.4.3. *For every $D \leq n$ and every k, there exists a graph G of diameter D such that any distributed algorithm that solves k-RW-SoD on G requires with high probability $\Omega(\min(\ell, D) + k)$ rounds.*

Proof. Consider a star graph of k branches and each branch has length ℓ. The lower bound of $\Omega(\min(\ell, D))$ rounds can be proved the same way as in Lemma 3.3.1 (by looking at the center node and any leaf node).

For the lower bound of $\Omega(k)$ rounds, let s be any neighbor of the center node and consider computing k walks of length 2 from s. With positive probability, there are $\Omega(k)$ different destinations. For s to know (and output) all these destination IDs, the algorithm needs $\Omega(k)$ rounds as the degree of s is just two. □

3.5 Better Bound when $\ell \geq t_{mix}$

In this section, we study the case when the length of the random walk sample is larger than the mixing time of the graph. This case is especially interesting in graphs with small mixing time, such as expanders and hypercubes (mixing time being $\tilde{O}(1)$). We show that for such graphs, random walks (both k-RW-DoS and k-RW-SoD) of length ℓ can be done more efficiently. In fact, we show a more general result: Random walks of length $\ell \geq t_{mix}$, where t_{mix} is the mixing time (the number of steps needed to be "close" to the stationary distribution, assuming that the graph is connected and non-bipartite) of an undirected unweighted graph G can be done in $O(D + k)$ rounds. Since mixing time depends on how close one wishes to be to the steady state distribution (there are various notions of mixing time, see e.g., [39]), we consider *approximate* sampling for this case. By this, we mean that we will sample nodes according to the steady state distribution which is *close* to the distribution after a walk of length ℓ when $\ell \geq t_{mix}$.

If one disregards the fact that this is an approximate sampling from the random walk, this improves the Few-Random-Walks algorithm (cf. Algorithm 10) in this special case. Note that in the rest of the chapter, we obtained *exact* random walk samples from the distribution of the corresponding length.

The rest of the section shows how one can sample a node according to the steady state distribution efficiently.

3.5.1 Algorithm

Our algorithm relies on the following observation.

Observation: *Approximately* sampling a node after a walk of length $\ell \geq t_{mix}$ only requires sampling a node v of degree d_v with probability $d_v/(2m)$.

This is due to the well-known fact that the distribution after a walk of length greater than the mixing time for undirected unweighted graphs is determined by the graph's degree distribution. In particular, the stationary probability of a node of degree d is $d/2m$ where m is the number of edges in the graph [123].

We now only need to show that sampling such a node can be done in $O(D + k)$ rounds. In fact, one can show something more general. One can sample a node from any fixed distribution in $O(D + k)$ rounds. We present the algorithm for this in Sample-Fixed-Distribution(s, H) (cf. Algorithm 12). The algorithm can be thought of as a modification of the first two sweeps of algorithm Sample-Destination (cf. Algorithm 15); this extension does not require any additional insight. The notation in the algorithm uses H to denote the distribution from which a node is to be sampled. Therefore, H_v is the probability with which the resulting node should be v. We state the algorithm more generality, where k nodes need to be sampled according to distribution H and the source needs to recover their IDs.

3.5.2 Analysis

We state the main result of this section below. The proof requires two parts - to verify the correctness of the sampling algorithm; and to bound the number of rounds.

Theorem 3.5.1. *Algorithm* Sample-Fixed-Distribution *(cf. Algorithm 12) solves k-RW-DoS and k-RW-SoD in $O(D+k)$ rounds for any $\ell \geq t_{mix}$.*

Proof. We first show that the nodes reaching the root are sampled from the true probability distribution H.

Consider any node b and let T be the subtree rooted at b. We claim that, for any node v in T, the probability v being in the first slot at b is exactly equal to $\frac{H_v}{H(b)}$ where $H(b) = \sum_{v \in T} H(v)$. We show this by induction. Our claim clearly holds for the base case where b is a leaf node. Now, for any non-leaf node b, assume that our claim is true for any children of b. That is, if b has j children and if a_i (for $1 \leq i \leq j$) is the node b receives from the i-th children, denoted by u_i, then a_i is picked from u_i with probability $\frac{H_{a_i}}{H(u_i)}$. Note from the algorithm that $p_0 = 1$ and $p_i = H(u_i$ for any $1 \leq i \leq j$. Therefore, for any i, the probability that a_i is picked by b is $\frac{H_{a_i}}{H(u_i)} \cdot \frac{H(u_i)}{H(b)} = \frac{H_{a_i}}{H(b)}$ as claimed.

Now, using the claim above, the probability of a node v being in the first slot of the root node is $\frac{H_v}{H(r)}$ where r is the root of the BFS tree of the entire graph. Since $H(r) = 1$, we have a sample from the required probability. This completes the proof of correctness.

Now we show that the number of rounds is $O(D+k)$. Constructing the BFS tree requires $O(D)$ rounds. The backward phase would require $O(D)$ rounds if only one token was being maintained, since the depth of the tree is $O(D)$. Now, with k tokens, observe that when a node receives tokens from all of its children it can immediately sample one token and forward it to its parent. In other word, it always sends messages to its parents once it receive messages from its children. Since each node receives only

k sets of messages, the number of rounds is $O(D + k)$. □

3.6 Using Routing Tables

We make a remark about the use of Get-More-Walks and Sample-Destination in our main algorithms including Single-Random-Walk. In Get-More-Walks, we perform η walks of length λ from a specified source, in λ rounds. At the end of this, all destinations are aware of the source, however, the source does not know the destinations ids. This is why Single-Random-Walk needs to invoke Sample-Destination later on. An alternative to this is to invest more rounds in Get-More-Walks as follows. The source can obtain all η destinations in $O(\eta + \lambda)$ rounds; this can be shown by a standard congestion + dilation argument. If this is done, then Sample-Destination is no longer required. The crucial point is that in the choice of our parameters, λ is more than η, and so the overall asymptotic bounds of the algorithm are not affected. Notice that we still need lD/λ rounds for the stitching phase (although we no longer need this for the Sample-Destination calls).

The advantage of getting rid of Sample-Destination is that the source node now does not need to construct BFS trees to obtain and sample from its destinations. If the nodes had access to a shortest path routing table, then the algorithm Single-Random-Walk will never need to construct BFS trees. While the construction of BFS tree in our algorithm is not a bottleneck in terms of number of rounds, this procedure is often unwieldy in practice. The use of routing tables instead, and simplifying the algorithm to not use Sample-Destination greatly increases the practicality of our method: no BFS tree construction is needed and the algorithm is very local and robust. Notice that the general idea of Sample-Destination is still required for our later results that use Sample-Fixed-Distribution.

3.7 Discussion

To conclude this chapter, recall that the main result here is a $\tilde{O}(\ell^{2/3}D^{1/3})$ algorithm for 1-RW-DoS which is later extended to k-RW-DoS algorithms (using $\tilde{O}((k\ell)^{2/3}D^{1/3})$ rounds for $k \leq \ell^2$ and $\tilde{O}(\sqrt{k\ell})$ rounds for $k \geq \ell^2$). We also consider other variations, special cases, and lower bounds.

This problem is still open, for both lower and upper bound. In particular, we conjecture that the true number of rounds for 1-RW-DoS is $\tilde{O}(\sqrt{\ell D})$. In the next chapter we resolve this conjecture. It will be also interesting to explore fast algorithms for performing random walks in directed graphs (both weighted and unweighted).

As noted earlier, the focus of this work is to improve the time complexity of random walks; however, this can come at the cost of increased message complexity. It would also be interesting to study tradeoffs between time and messages.

Algorithm 9 SAMPLE-DESTINATION(v)

Input: Starting node v, and desired walk length λ.
Output: A node sampled from among the stored λ-length walks from v.

Sweep 1: (Perform BFS tree)

1: Construct a Breadth-First-Search (BFS) tree rooted at v. While constructing, every node stores its parent's ID. Denote such tree by T.

Sweep 2: (Tokens travel up the tree, sampling as you go)

1: We divide T naturally into levels 0 through D (where nodes in level D are leaf nodes and the root node s is in level 0).
2: Tokens are held by nodes as a result of doing walks of length λ from v (which is done in either Phase 1 or GET-MORE-WALKS (cf. Algorithm 14)) A node could have more than one token.
3: Every node u that holds token(s) picks one token, denoted by d_0, uniformly at random and lets c_0 denote the number of tokens it has.
4: **for** $i = D$ down to 0 **do**
5: Every node u in level i that either receives token(s) from children or possesses token(s) itself do the following.
6: Let u have tokens $d_0, d_1, d_2, \ldots, d_q$, with counts $c_0, c_1, c_2, \ldots, c_q$ (including its own tokens). The node v samples one of d_0 through d_q, with probabilities proportional to the respective counts. That is, for any $1 \leq j \leq q$, d_j is sampled with probability $\frac{c_j}{c_0+c_1+\ldots+c_q}$.
7: The sampled token is sent to the parent node (unless already at root), along with a count of $c_0 + c_1 + \ldots + c_q$ (the count represents the number of tokens from which this token has been sampled).
8: **end for**
9: The root output the ID of the owner of the final sampled token. Denote such node by u_d.

Sweep 3: (Go and delete the sampled destination)

1: v sends a message to u_d (in D rounds through graph edges). u_d deletes one token of v it is holding (so that this random walk of length λ is not reused/re-stitched).

Algorithm 10 FEW-RANDOM-WALKS($\{s_j\}, 1 \leq j \leq k$, ℓ)

Input: Starting nodes $s_1, s_2, \ldots, s_k$ (not necessarily distinct), and desired walks of length ℓ.
Output: Each destination outputs the ID of its corresponding source.

Phase 1: (Each node performs η random walks of length λ)

1: Perform η walks of length λ as in Phase 1 of algorithm SINGLE-RANDOM-WALK (cf. Algorithm 13).

Phase 2: (Stitch ℓ/λ short walks)

1: **for** $j = 1$ to k **do**
2: Consider source s_j. Use algorithm SINGLE-RANDOM-WALK to perform a walk of length ℓ from s_j.
3: When algorithm SINGLE-RANDOM-WALK terminates, the sampled destination outputs ID of the source s_j.
4: **end for**

Algorithm 11 MANY-RANDOM-WALKS($(s_1, k_1), (s_2, k_2), \ldots, (s_q, k_q)$ ℓ)

Input: Desired walk length l, q distinct sources $s_1, \ldots, s_q$ and number of walks k_i for each source s_i where $\sum_{i=1}^{q} k_i = k$
Output: For each i, destinations of k walks of length ℓ starting from s_i.

Phase 1: Each node performs η random walks of length ℓ. See phase 1 of SINGLE-RANDOM-WALK (cf. Algorithm 13).
Phase 2:

1: **for** $i = 1$ to q **do**
2: **if** $k_i > \eta$ **then**
3: s_i calls GET-MORE-WALK(s_i, $k_i - \eta$, ℓ).
4: **end if**
5: **end for**

Algorithm 12 SAMPLE-FIXED-DISTRIBUTION $(\{s_1, s_2, ..., s_k\}, k, H)$

Input: Source nodes $s_1, s_2, ..., s_k$, number of destinations to be sampled, k, and the distribution to be sampled from, H.

Output: k sampled destinations according to the distribution H.

1: Let r be any node (can be chosen by a leader election algorithm). Construct a BFS tree rooted at r. While constructing, every node stores its parent node/edge.

2: Every source node sends their IDs to r via the BFS tree. r now samples k destinations as follows.

3: k tokens are passed from leaves of the BFS eventually to r. The BFS is divided into D levels. Initially, each leaf node (level D node) fills all the k slots with its node ID and sends it to its parent. It also sends a value to reflect the sum of probabilities of all nodes in the subtree rooted at itself, according to the distribution H. For a leaf node, the value is its own probability in H.

4: **for** $x = D - 1$ to 1 **do**

5: When a node v in level x receives one or more sets of tokens and values from its children, it does the following for each token slot. For the first slot, suppose that it receives node IDs $a_1, a_2, \ldots, a_j$ with values $p_1, p_2, \ldots, p_j$. Let a_0 denote its own ID and p_0 denote its distribution according to H. The node v samples one of a_0 through a_j, with probabilities proportional to the values; i.e., for any $i \leq j$, the node a_i is picked with probability $p_i/(p_0 + p_1 + ... + p_j)$.

6: The above is done for each of the k slots. The node v then updates the value to $p_1 + p_2 + \ldots + p_j + p_0$. These k slots and the value are then sent by v to its parent (unless v is the root).

7: **end for**

8: Finally, r receives all destinations (IDs in the k slots). It randomly matches destinations with the sources. It then sends each destination ID to the corresponding source.

CHAPTER IV

IMPROVED RANDOM WALK ALGORITHMS AND APPLICATIONS IN DISTRIBUTED NETWORKS

In this chapter, we again focus on the problem of performing random walks efficiently in a distributed network. Given bandwidth constraints, the goal is to minimize the number of rounds required to obtain a random walk sample. We [52] first present a fast sublinear time distributed algorithm for performing random walks whose time complexity is sublinear in the length of the walk. Our algorithm performs a random walk of length ℓ in $\tilde{O}(\sqrt{\ell D})$ rounds (with high probability) on an undirected network, where D is the diameter of the network. This improves over the previous best algorithm that ran in $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds presented in the previous chapter. We further extend our algorithms to efficiently perform k independent random walks in $\tilde{O}(\sqrt{k\ell D}+k)$ rounds. We then show that there is a fundamental difficulty in improving the dependence on ℓ any further by proving a lower bound of $\Omega(\sqrt{\frac{\ell}{\log \ell}}+D)$ under a general model of distributed random walk algorithms. Our random walk algorithms are useful in speeding up distributed algorithms for a variety of applications that use random walks as a subroutine. We present two main applications. First, we give a fast distributed algorithm for computing a random spanning tree (RST) in an arbitrary (undirected) network which runs in $\tilde{O}(\sqrt{m}D)$ rounds (with high probability; here m is the number of edges). Our second application is a fast decentralized algorithm for estimating mixing time and related parameters of the underlying network. Our algorithm is fully decentralized and can serve as a building block in the design of topologically-aware networks.

4.1 Related Work and Contributions

As stated previously, random walks play a central role in computer science, spanning a wide range of areas in both theory and practice. The focus of this chapter is also random walks in networks, in particular, decentralized algorithms for performing random walks in arbitrary networks. Random walks are used as an integral subroutine in a wide variety of network applications ranging from token management and load balancing to search, routing, information propagation and gathering, network topology construction and building random spanning trees (as mentioned in the previous chapter and see the corresponding paper [51] and the references therein).

In this chapter, we present a sublinear time (sublinear in ℓ) distributed random walk sampling algorithm that is significantly faster than the previous best result. Our algorithm runs in time $\tilde{O}(\sqrt{\ell D})$ rounds. We then present an almost matching lower bound that applies to a general class of distributed algorithms (our algorithm also falls in this class). Finally, we present two key applications of our algorithm. The first is a fast distributed algorithm for computing a random spanning tree, a fundamental spanning tree problem that has been studied widely in the classical setting (see e.g., [95] and references therein). To the best of our knowledge, our algorithm gives the fastest known running time in an arbitrary network. The second is to devising efficient decentralized algorithms for computing key global metrics of the underlying network — mixing time, spectral gap, and conductance. Such algorithms can be useful building blocks in the design of *topologically (self-)aware* networks, i.e., networks that can monitor and regulate themselves in a decentralized fashion. For example, efficiently computing the mixing time or the spectral gap, allows the network to monitor connectivity and expansion properties of the network.

4.1.1 Distributed Computing Model

Recall the CONGEST model as follows. Consider an undirected, unweighted, connected n-node graph $G = (V, E)$. Assume that each node is associated with a distinct identity number from the set $\{1, 2, ..., n\}$. At the beginning of the computation, each node v accepts as input its own identity number and the identity numbers of its neighbors in G. The nodes are allowed to communicate through the edges of the graph G. The communication is synchronous, and occurs in discrete pulses, called *rounds*. In each round each node v is allowed to send an arbitrary message of size $O(\log n)$ through each edge $e = (v, u)$ that is adjacent to v, and the message will arrive to u at the end of the current round.

There are several measures of efficiency of distributed algorithms, but as in the last chapter we will concentrate on one of them, specifically, *the running time*, that is, the number of rounds of distributed communication. (Note that the computation that is performed by the nodes locally is "free", i.e., it does not affect the number of rounds.) Many fundamental network problems such as minimum spanning tree, shortest paths, etc. have been addressed in this model (e.g., see [115, 131, 128]). In particular, there has been much research into designing very fast distributed approximation algorithms (that are even faster at the cost of producing sub-optimal solutions) for many of these problems (see e.g., [60, 59, 99, 98]). Such algorithms can be useful for large-scale resource-constrained and dynamic networks where running time is crucial.

4.1.2 Problem Statement, Motivation, and Related Work

The basic problem we address is the following. We are given an arbitrary undirected, unweighted, and connected n–node network $G = (V, E)$ and a (source) node $s \in V$. The goal is to devise a distributed algorithm such that, in the end, s outputs the ID of a node v which is randomly picked according to the probability that it is the destination of a random walk of length ℓ starting at s. Throughout this chapter as

well, we assume the standard (simple) random walk: in each step, an edge is taken from the current node x with probability proportional to $1/d(x)$ where $d(x)$ is the degree of x. Our goal is to output a true random sample from the ℓ-walk distribution starting from s.

For clarity, observe that the following naive algorithm solves the above problem in $O(\ell)$ rounds: The walk of length ℓ is performed by sending a token for ℓ steps, picking a random neighbor with each step. Then, the destination node v of this walk sends its ID back (along the same path) to the source for output. Our goal is to perform such sampling with significantly less number of rounds, i.e., in time that is sublinear in ℓ. On the other hand, we note that it can take too much time (as much as $\Theta(|E| + D)$ time) in the CONGEST model to collect all the topological information at the source node (and then computing the walk locally).

This problem was proposed in [51] under the name *Computing One Random Walk where Source Outputs Destination (1-RW-SoD)* (for short, this problem will be simply called *Single Random Walk* in this work), wherein the first sublinear time distributed algorithm was provided, requiring $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds ($\tilde{O}$ hides polylog(n) factors); this improves over the naive $O(\ell)$ algorithm when the walk is long compared to the diameter (i.e., $\ell = \Omega(D \operatorname{polylog} n)$ where D is the diameter of the network). This was the first result to break past the inherent sequential nature of random walks and beat the naive ℓ round approach, despite the fact that random walks have been used in distributed networks for long and in a wide variety of applications.

There are two key motivations for obtaining sublinear time bounds. The first is that in many algorithmic applications, walks of length significantly greater than the network diameter are needed. For example, this is necessary in both the applications presented later in the chapter, namely distributed computation of a random spanning tree (RST) and computation of mixing time. In the RST algorithm, we need to perform a random walk of expected length $O(mD)$ (where m is the number of edges

in the network). In decentralized computation of mixing time, we need to perform walks of length at least equal to the mixing time which can be significantly larger than the diameter (e.g., in a random geometric graph model [126], a popular model for ad hoc networks, the mixing time can be larger than the diameter by a factor of $\Omega(\sqrt{n})$.) More generally, many real-world communication networks (e.g., ad hoc networks and peer-to-peer networks) have relatively small diameter, and random walks of length at least the diameter are usually performed for many sampling applications, i.e., $\ell >> D$. It should be noted that if the network is rapidly mixing/expanding which is sometimes the case in practice, then sampling from walks of length $\ell >> D$ is close to sampling from the steady state (degree) distribution; this can be done in $O(D)$ rounds (note however, that this gives only an approximately close sample, not the exact sample for that length). However, such an approach fails when ℓ is smaller than the mixing time.

The second motivation is understanding the time complexity of distributed random walks. Random walk is essentially a global problem which requires the algorithm to "traverse" the entire network. Classical "global" problems include the minimum spanning tree, shortest path etc. Network diameter is an inherent lower bound for such problems. Problems of this type raise the basic question whether n (or ℓ as the case here) time is essential or is the network diameter D, the inherent parameter. As pointed out in the seminal work of [69], in the latter case, it would be desirable to design algorithms that have a better complexity for graphs with low diameter.

The high-level idea used in the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm in [51] is to "prepare" a few short walks in the beginning (executed in parallel) and then carefully stitch these walks together later as necessary. The same general approach was introduced in [48] to find random walks in data streams with the main motivation of finding PageRank. However, the two models have very different constraints and motivations and hence the subsequent techniques used in [51] and [48] are very different.

Recently, Sami and Twigg [133] consider lower bounds on the communication complexity of computing stationary distribution of random walks in a network. Although, their problem is related to our problem, the lower bounds obtained do not imply anything in our setting. Other recent works involving multiple random walks in different settings include Alon et. al. [9], and Cooper et al. [43].

4.1.3 Our Results

- **A Fast Distributed Random Walk Algorithm:** We present a sublinear, almost time-optimal, distributed algorithm for the single random walk problem in arbitrary networks that runs in time $\tilde{O}(\sqrt{\ell D})$, where ℓ is the length of the walk (cf. Section 4.2). This is a significant improvement over the naive ℓ-round algorithm for $\ell = \Omega(D)$ as well as over the previous best running time of $\tilde{O}(\ell^{2/3}D^{1/3})$ [51]. The dependence on ℓ is reduced from $\ell^{2/3}$ to $\ell^{1/2}$.

  Our algorithm in this chapter uses an approach similar to that of [51] but exploits certain key properties of random walks to design an even faster sublinear time algorithm. Our algorithm is randomized (Las Vegas type, i.e., it always outputs the correct result, but the running time claimed is with high probability) and is conceptually simpler compared to the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm (whose running time is deterministic). While the previous (slower) algorithm [51] applies to the more general Metropolis-Hastings walk, in this work we focus primarily on the simple random walk for the sake of obtaining the best possible bounds in this commonly used setting.

  One of the key ingredients in the improved algorithm is proving a bound on the number of times any node is visited in an ℓ-length walk, for any length $\ell = O(m^2)$. We show that w.h.p. any node x is visited at most $\tilde{O}(d(x)\sqrt{\ell})$ times, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x). We then show that if only certain ℓ/λ special points of the walk (called as

connector points) are observed, then any node is observed only $\tilde{O}(d(x)\sqrt{\ell}/\lambda)$ times. The algorithm starts with all nodes performing short walks (of length uniformly random in the range λ to 2λ for appropriately chosen λ) efficiently simultaneously; here the randomly chosen lengths play a crucial role in arguing about a suitable spread of the connector points. Subsequently, the algorithm begins at the source and carefully stitches these walks together till ℓ steps are completed.

We also extend to give algorithms for computing k random walks (from any k sources —not necessarily distinct) in $\tilde{O}\left(\min(\sqrt{k\ell D}+k, k+\ell)\right)$ rounds. Computing k random walks is useful in many applications such as the one we present below on decentralized computation of mixing time and related parameters. While the main requirement of our algorithms is to just obtain the random walk samples (i.e. the end point of the ℓ step walk), our algorithms can regenerate the entire walks such that each node knows its position(s) among the ℓ steps. Our algorithm can be extended to do this in the same number of rounds.

- **A Lower Bound:** We establish an almost matching lower bound on the running time of distributed random walk that applies to a general class of distributed random walk algorithms. We show that any algorithm belonging to the class needs at least $\Omega(\sqrt{\frac{\ell}{\log \ell}}+D)$ rounds to perform a random walk of length ℓ; notice that this lower bound is nontrivial even in graphs of small ($D = O(\log n)$) diameter (cf. Section 4.3). Broadly speaking, we consider a class of token forwarding-type algorithms where nodes can only store and (selectively) forward tokens (here tokens are $O(\log n)$-sized messages consisting of two node ids identifying the beginning and end of a segment — we make this more precise in Section 4.3). Selective forwarding (more general than just store and forwarding) means that nodes can omit to forward certain segments (to reduce number of messages), but they cannot alter tokens in any way (e.g., resort

to data compression techniques). This class includes many natural algorithms, including the algorithm in this chapter.

Our technique involves showing the same non-trivial lower bound for a problem that we call *path verification*. This simpler problem appears quite basic and can have other applications. Informally, given a graph G and a sequence of ℓ vertices in the graph, the problem is for some (source) node in the graph to verify that the sequence forms a path. One main idea in this proof is to show that independent nodes may be able to verify short *local* paths; however, to be able to *merge* these together and verify an ℓ-length path would require exchanging several messages. The trade-off is between the lengths of the local paths that are verified and the number of such local paths that need to be combined. Locally verified paths can be exchanged in one round, and messages can be exchanged at all nodes. Despite this, we show that the bandwidth restriction necessitates a large number of rounds even if the diameter is small. We then show a reduction to the random walk problem, where we require that each node in the walk should know its (correct) position(s) in the walk.

Similar non-trivial matching lower bounds on running time are known only for a few important problems in distributed computing, notably the minimum spanning tree problem (e.g., see [130, 61]). Peleg and Rabinovich [130] showed that $\tilde{\Omega}(\sqrt{n})$ time is required for constructing an MST even on graphs of small diameter (for any $D = \Omega(\log n)$) and [103] showed an essentially matching upper bound.

- **Applications:** Our faster distributed random walk algorithm can be used in speeding up distributed applications where random walks arise as a subroutine. Such applications include distributed construction of expander graphs, checking whether a graph is an expander, construction of random spanning trees, and

random-walk based search (we refer to [51] for details). Here, we present two key applications:

(1) *A Fast Distributed Algorithm for Random Spanning Trees (RST):* We give a $\tilde{O}(\sqrt{m}D)$ time distributed algorithm (cf. Section 4.4.1) for uniformly sampling a random spanning tree in an arbitrary undirected (unweighted) graph (i.e., each spanning tree in the underlying network has the same probability of being selected). (m denotes the number of edges in the graph.) Spanning trees are fundamental network primitives and distributed algorithms for various types of spanning trees such as minimum spanning tree (MST), breadth-first spanning tree (BFS), shortest path tree, shallow-light trees etc., have been studied extensively in the literature [131]. However, not much is known about the distributed complexity of the random spanning tree problem. The centralized case has been studied for many decades, see e.g., the recent work of [95] and the references therein; also see the recent work of Goyal et al. [77] which gives nice applications of RST to fault-tolerant routing and constructing expanders. In the distributed context, the work of Bar-Ilan and Zernik [17] give a distributed RST algorithm for two special cases, namely that of a complete graph (running in constant time) and a synchronous ring (running in $O(n)$ time). The work of [16] give a self-stablizing distributed algorithm for constructing a RST in a wireless ad hoc network and mentions that RST is more resilient to transient failures that occur in mobile ad hoc networks.

Our algorithm works by giving an efficient distributed implementation of the well-known Aldous-Broder random walk algorithm [5, 32] for constructing a RST.

(2) *Decentralized Computation of Mixing Time.* We present a fast decentralized algorithm for estimating mixing time, conductance and spectral gap of the network (cf. 4.4.2). In particular, we show that given a starting point x, the mixing

time with respect to x, called τ^x_{mix}, can be estimated in $\tilde{O}(n^{1/2} + n^{1/4}\sqrt{D\tau^x_{mix}})$ rounds. This gives an alternative algorithm to the only previously known approach by Kempe and McSherry [96] that can be used to estimate τ^x_{mix} in $\tilde{O}(\tau^x_{mix})$ rounds.[1] To compare, we note that when $\tau^x_{mix} = \omega(n^{1/2})$ the present algorithm is faster (assuming D is not too large).

The work of [71] discusses spectral algorithms for enhancing the topology awareness, e.g., by identifying and assigning weights to critical links. However, the algorithms are centralized, and it is mentioned that obtaining efficient decentralized algorithms is a major open problem. Our algorithms are fully decentralized and based on performing random walks, and so more amenable to dynamic and self-organizing networks.

4.2 A Sublinear Time Distributed Random Walk Algorithm

4.2.1 Description of the Algorithm

We first describe the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm in [51] and then highlight the changes in our current algorithm. The current algorithm is randomized and uses several new ideas that are crucial in obtaining the new bound.

The high-level idea is to perform "many" short random walks in parallel and later stitch them together as needed (see Figure 2 in Section 5.5). In the first phase of the algorithm Single-Random-Walk (we refer to Section 5.5 for pseudocodes of all algorithms and subroutines), each node performs η independent random walks of length λ. (Only the destination of each of these walks is aware of its source, but the sources do not know destinations right away.) It is shown that this takes $\tilde{O}(\eta\lambda)$ rounds with high probability. Subsequently, the source node that requires a walk of

[1] Note that [96] in fact do more and give a decentralized algorithm for computing the top k eigenvectors of a weighted adjacency matrix that runs in $O(\tau_{mix}\log^2 n)$ rounds if two adjacent nodes are allowed to exchange $O(k^3)$ messages per round, where τ_{mix} is the mixing time and n is the size of the network.

length ℓ extends a walk of length λ by "stitching" walks. If the end point of the first λ length walk is u, one of u's λ length walks is used to extend. When at u, one of its λ-length walk destinations are sampled uniformly (to preserve randomness) using Sample-Destination in $O(D)$ rounds. (We call such u and other nodes at the stitching points as *connectors* — cf. Algorithm 1.) Each stitch takes $O(D)$ rounds (via the shortest path). This process is extended as long as unused λ-length walks are available from visited nodes. If the walk reaches a node v where all η walks have been used up (which is a key difficulty), then Get-More-Walks is invoked. Get-More-Walks performs η more walks of length λ from v, and this can be done in $\tilde{O}(\lambda)$ rounds. The number of times Get-More-Walks is invoked can be bounded by $\frac{\ell}{\eta\lambda}$ in the worst case by an amortization argument. The overall bound on the algorithm is $O(\eta\lambda + \ell D/\lambda + \frac{\ell}{\eta})$. The bound of $\tilde{O}(\ell^{2/3}D^{1/3})$ follows from appropriate choice of parameters η and λ.

The current algorithm uses two crucial ideas to improve the running time. The first idea is to bound the number of times any node is visited in a random walk of length ℓ (in other words, the number of times Get-More-Walks is invoked). Instead of the worst case analysis in [51], the new bound is obtained by bounding the number of times any node is visited (with high probability) in a random walk of length ℓ on an undirected unweighted graph. The number of visits to a node beyond the mixing time can be bounded using its stationary probability distribution. However, we need a bound on the visits to a node for any ℓ-length walk starting from the first step. We show a somewhat surprising bound that applies to an ℓ-length (for $\ell = O(m^2)$) random walk on any arbitrary (undirected) graph: *no node x is visited more than $\tilde{O}(d(x)\sqrt{\ell})$ times*, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x) (cf. Lemma 4.2.6). Note that this bound does not depend on any other parameter of the graph, just on the (local) degree of the node and the length of the walk. This bound is tight in general (e.g., consider a line and a walk of length n).

The above bound is not enough to get the desired running time, as it does not say anything about the distribution of connectors when we chop the length ℓ walk into ℓ/λ pieces. We have to bound the number of visits to a node as a connector in order to bound the number of times Get-More-Walks is invoked. To overcome this we use a second idea: Instead of nodes performing walks of length λ, each such walk i is of length $\lambda + r_i$ where r_i is a random number in the range $[0, \lambda - 1]$. Notice that the random numbers are independent for each walk. We show the following "uniformity lemma": if the short walks are now of a random length in the range of $[\lambda, 2\lambda - 1]$, then if a node u is visited at most N_u times in an ℓ step walk, then the node is visited at most $\tilde{O}(N_u/\lambda)$ times as an endpoint of a short walk (cf. Lemma 4.2.7). This modification to Single-Random-Walk allows us to bound the number of visits to each node (cf. Lemma 4.2.7).

The change of the short walk length above leads to two modifications in Phase 1 of Single-Random-Walk and Get-More-Walks. In Phase 1, generating η walks of different lengths from each node is straightforward: Each node simply sends η tokens containing the source ID and the desired length. The nodes keep forwarding these tokens with decreased desired walk length until the desired length becomes zero. The modification of Get-More-Walks is tricker. To avoid congestion, we use the idea of *reservoir sampling* [138]. In particular, we add the following process at the end of the Get-More-Walks algorithm in [51]:

for $i = 0$ to $\lambda - 1$ **do**

For each message, independently with probability $\frac{1}{\lambda - i}$, stop sending the message further and save the ID of the source node (in this event, the node with the message is the destination). For messages M that are not stopped, each node picks a neighbor correspondingly and sends the messages forward as before.

end for

The reason it needs to be done this way is that if we first sampled the walk length r, independently for each walk, in the range $[0, \lambda - 1]$ and then extended each walk accordingly, the algorithm would need to pass r independently for each walk. This will cause congestion along the edges; no congestion occurs in the mentioned algorithm as only the *count* of the number of walks along an edge are passed to the node across the edge. Therefore, we need to decide when to stop on the fly using reservoir sampling.

We also have to make another modification in Phase 1 due to the new bound on the number of visits. Recall that, in this phase, each node prepares η walks of length λ. However, since the new bound of visits of each node x is proportional to its degree $d(x)$ (see Lemma 4.2.6), we make each node prepare $\eta d(x)$ walks instead. We show that Phase 1 uses $\tilde{O}(\eta\lambda)$ rounds, instead of $\tilde{O}(\frac{\lambda\eta}{\delta})$ rounds where δ is the minimum degree in the graph (cf. Lemma 4.2.3).

To summarize, the main algorithm for performing a single random walk is Single-Random-Walk. This algorithm, in turn, uses Get-More-Walks and Sample-Destination. The key modification is that, instead of creating short walks of length λ each, we create short walks where each walk has length in range $[\lambda, 2\lambda - 1]$. To do this, we modify the Phase 1 of Single-Random-Walk and Get-More-Walks.

We now state four lemmas which are similar to the Lemma 2.2-2.6 in [51]. However, since the algorithm here is a modification of that in [51], we include the full proofs in Section 4.6.1.2.

Lemma 4.2.1. *Phase 1 finishes in $O(\lambda\eta \log n)$ rounds with high probability.*

Lemma 4.2.2. *For any v,* Get-More-Walks(v, η, λ) *always finishes within $O(\lambda)$ rounds.*

Lemma 4.2.3. Sample-Destination *always finishes within $O(D)$ rounds.*

Lemma 4.2.4. *Algorithm* Sample-Destination*(v) (cf. Algorithm 15) returns a destination from a random walk whose length is uniform in the range $[\lambda, 2\lambda - 1]$.*

4.2.2 Analysis

The following theorem states the main result of this Section. It states that the algorithm Single-Random-Walk correctly samples a node after a random walk of ℓ steps and the algorithm takes, with high probability, $\tilde{O}\left(\sqrt{\ell D}\right)$ rounds where D is the diameter of the graph. Throughout this section, we assume that ℓ is $O(m^2)$, where m is the number of edges in the network. If ℓ is $\Omega(m^2)$, the required bound is easily achieved by aggregating the graph topology (via upcast) onto one node in $O(m + D)$ rounds (e.g., see [131]). The difficulty lies in proving for $\ell = O(m^2)$.

Theorem 4.2.5. *For any ℓ, Algorithm* Single-Random-Walk *(cf. Algorithm 13) solves* 1*-RW-DoS (the Single Random Walk Problem) and, with probability at least $1 - \frac{2}{n}$, finishes in $\tilde{O}\left(\sqrt{\ell D}\right)$ rounds.*

We prove the above theorem using the following lemmas. As mentioned earlier, to bound the number of times Get-More-Walks is invoked, we need a technical result on random walks that bounds the number of times a node will be visited in a ℓ-length random walk. Consider a simple random walk on a connected undirected graph on n vertices. Let $d(x)$ denote the degree of x, and let m denote the number of edges. Let $N_t^x(y)$ denote the number of visits to vertex y by time t, given the walk started at vertex x. Now, consider k walks, each of length ℓ, starting from (not necessary distinct) nodes $x_1, x_2, \ldots, x_k$. We show a key technical lemma (proof in Section 4.6.1.4) that applies to a random walk on any graph: With high probability, no vertex y is visited more than $24d(x)\sqrt{k\ell + 1}\log n + k$ times.

Lemma 4.2.6. *For any nodes $x_1, x_2, \ldots, x_k$, and $\ell = O(m^2)$,*

$$\Pr\Big(\exists y \ s.t. \ \sum_{i=1}^{k} N_\ell^{x_i}(y) \geq 24d(x)\sqrt{k\ell + 1}\log n + k\Big) \leq 1/n\,.$$

This lemma says that the number of visits to each node can be bounded. However, for each node, we are only interested in the case where it is used as a connector. The

lemma below shows that the number of visits as a connector can be bounded as well; i.e., if any node v_i appears t times in the walk, then it is likely to appear roughly t/λ times as connectors.

Lemma 4.2.7. *For any vertex v, if v appears in the walk at most t times then it appears as a connector node at most $t(\log n)^2/\lambda$ times with probability at least $1-1/n^2$.*

Intuitively, this argument is simple, since the connectors are spread out in steps of length approximately λ. However, there might be some *periodicity* that results in the same node being visited multiple times but *exactly* at λ-intervals. This is where we crucially use the fact that the algorithm uses walks of length $\lambda + r$ where r is chosen uniformly at random from $[0, \lambda - 1]$. The proof then goes via constructing another process equivalent to partitioning the ℓ steps in to intervals of λ and then sampling points from each interval. We analyze this by carefully constructing a different process that stochastically dominates the process of a node occurring as a connector at various steps in the ℓ-length walk and then use a Chernoff bound argument. The detailed proof is presented in Section 4.6.1.3.

Now we are ready to prove Theorem 4.2.5.

Proof of Theorem 4.2.5. First, we claim, using Lemma 4.2.6 and 4.2.7, that each node is used as a connector node at most $\frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times with probability at least $1 - 2/n$. To see this, observe that the claim holds if each node x is visited at most $t(x) = 24d(x)\sqrt{\ell + 1}\log n$ times and consequently appears as a connector node at most $t(x)(\log n)^2/\lambda$ times. By Lemma 4.2.6, the first condition holds with probability at least $1 - 1/n$. By Lemma 4.2.7 and the union bound over all nodes, the second condition holds with probability at least $1 - 1/n$, provided that the first condition holds. Therefore, both conditions hold together with probability at least $1 - 2/n$ as claimed.

Now, we choose $\eta = 1$ and $\lambda = 24\sqrt{\ell D}(\log n)^3$. By Lemma 4.2.1, Phase 1 finishes in $\tilde{O}(\lambda\eta) = \tilde{O}(\sqrt{\ell D})$ rounds with high probability. For Phase 2, Sample-Destination is invoked $O(\frac{\ell}{\lambda})$ times (only when we stitch the walks) and therefore, by Lemma 4.2.3, contributes $O(\frac{\ell D}{\lambda}) = \tilde{O}(\sqrt{\ell D})$ rounds. Finally, we claim that Get-More-Walks is never invoked, with probability at least $1 - 2/n$. To see this, recall our claim above that each node is used as a connector node at most $\frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times. Moreover, observe that we have prepared this many walks in Phase 1; i.e., after Phase 1, each node has $\eta\lambda d(x) = \frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ short walks. The claim follows.

Therefore, with probability at least $1-2/n$, the rounds are $\tilde{O}(\sqrt{\ell D})$ as claimed. □

Regenerating the entire random walk: It is important to note that our algorithm can be extended to regenerate the entire walk. As described above, the source node obtains the sample after a random walk of length ℓ. In certain applications, it may be desired that the entire random walk be obtained, i.e., every node in the ℓ length walk knows its position(s) in the walk. This can be done by first informing all intermediate connecting nodes of their position (since there are only $O(\sqrt{\ell})$ such nodes). Then, these nodes can regenerate their $O(\sqrt{\ell})$ length short walks; this can be completed in $\tilde{O}(\sqrt{\ell D})$ rounds without congestion, with high probability.

4.2.3 Extension to Computing k Random Walks

We now consider the scenario when we want to compute k walks of length ℓ from different (not necessary distinct) sources $s_1, s_2, \ldots, s_k$. We show that Single-Random-Walk can be extended to solve this problem. Consider the following algorithm.

Many-Random-Walks: Let $\lambda = (24\sqrt{k\ell D + 1}\log n + k)(\log n)^2$ and $\eta = 1$. If $\lambda > \ell$ then run the naive random walk algorithm, i.e., the sources find walks of length ℓ simultaneously by sending tokens. Otherwise, do the following. First, modify Phase 2 of Single-Random-Walk to create multiple walks, one at a time; i.e., in

the second phase, we stitch the short walks together to get a walk of length ℓ starting at s_1 then do the same thing for s_2, s_3, and so on. We state the theorem below and the proof is placed in Section 4.6.1.5.

Theorem 4.2.8. MANY-RANDOM-WALKS *finishes in* $\tilde{O}\left(\min(\sqrt{k\ell D}+k, k+\ell)\right)$ *rounds with high probability.*

4.3 Lower bound

In this section, we show an almost tight lower bound on the time complexity of performing a distributed random walk. At the end of the walk, we require that each node in the walk should know its correct position(s) among the ℓ steps. We show that any distributed algorithm needs at least $\Omega\left(\sqrt{\frac{\ell}{\log \ell}}\right)$ rounds, even in graphs with low diameter. Note that $\Omega(D)$ is a lower bound [51]. Also note that if a source node wants to sample k destinations from independent random walks, then $\Omega(k)$ is also a lower bound as the source may need to receive $\Omega(k)$ distinct messages. Therefore, for k walks, the lower bound we show is $\Omega(\sqrt{\frac{\ell}{\log \ell}}+k+D)$ rounds. (The rest of the section omits the $\Omega(k+D)$ term.) In particular, we show that there exists a n-node graph of diameter $O(\log n)$ such that any distributed algorithm needs at least $\Omega(\sqrt{\frac{n}{\log n}})$ time to perform a walk of length n. Our lower bound proof makes use of a lower bound for another problem that we call as the *Path Verification problem* defined as follows. Informally, the Path Verification problem is for some node v to verify that a given sequence of nodes in the graph is a valid path of length ℓ.

Definition 4.3.1 (PATH-VERIFICATION Problem)**.** *The input of the problem consists of an integer ℓ, a graph $G = (V, E)$, and ℓ nodes $v_1, v_2, ..., v_\ell$ in G. To be precise, each node v_i initially has its order number i.*

The goal is for some node v to "verify" that the above sequence of vertices forms an ℓ-length path, i.e., if (v_i, v_{i+1}) forms an edge for all $1 \leq i \leq \ell - 1$. Specifically, v should output "yes" if the sequence forms an ℓ-length path and "no" otherwise.

We show a lower bound for the Path Verification problem that applies to a very general class of verification algorithms defined as follows. Each node can (only) verify a segment of the path that it knows either directly or indirectly (by learning form its neighbors), as follows. Initially each node knows only the trivial segment (i.e. the vertex itself). If a vertex obtains from its neighbor a segment $[i_1, j_1]$ and it has already verified segment $[i_2, j_2]$ that overlaps with $[i_1, j_1]$ (say, $i_1 < i_2 < j_1 < j_2$) then it can verify a larger interval ($[i_1, j_2]$). Note that a node needs to only send the endpoints of the interval that it already verifies (hence larger intervals are better). (See Figure 1 in Section 5.5 for an example.) The goal of the problem is that, in the end, some node verifies the entire segment $[1, \ell]$. We would like to determine a lower bound for the running time of any distributed algorithm for the above problem.

A lower bound for the Path Verification problem, implies a lower bound for the random walk problem as well. The reason is as follows. Both problems involve constructing a path of some specified length ℓ. Intuitively, the former is a simpler problem, since we are not verifying whether the local steps are chosen randomly, but just whether the path is valid and is of length ℓ. On the other hand, any algorithm for the random walk problem (including our algorithm of Section 4.2), also solves the Path Verification problem, since the path it constructs should be a valid path of length ℓ. It is straightforward to make any distributed algorithm that computes a random walk to also verify that indeed the random walk is a valid walk of appropriate length. This is essential for correctness, as otherwise, an adversary can always change simply one edge of the graph and ensure that the walk is wrong.

In the next section we first prove a lower bound for the Path Verification problem. Then we show the same lower bound holds for the random walk problem by giving a reduction.

4.3.1 Lower Bound for the Path Verification Problem

The main result of this section is the following theorem.

Theorem 4.3.2. *For every n, and $\ell \leq n$ there exists a graph G_n of $\Theta(n)$ vertices and diameter $O(\log n)$, and a path P of length ℓ such that any algorithm that solves the* Path-Verification *problem on G_n and P requires more than k rounds, where $k = \sqrt{\frac{\ell}{\log \ell}}$.*

The rest of the section is devoted to proving the above Theorem. We start by defining G_n.

Definition 4.3.3 (Graph G_n)**.** *Let k' be an integer such that k is a power of 2 and $k'/2 \leq 4k < k'$. Let n' be such that $n' \geq n$ and k' divides n'. We construct G_n having $(n' + 2k' - 1) = O(n)$ nodes as follows. First, we construct a path $P = v_1 v_2 ... v_{n'}$. Second, we construct a binary T having k' leaf nodes. Let $u_1, u_2, ..., u_{k'}$ be its leaves from left to right. Finally, we connect P with T by adding an edge $u_i v_{jk'+i}$ for every i and j. We will denote the root of T by x and its left and right children by l and r respectively. Clearly, G_n has diameter $O(\log n)$. We then consider a path of length $\ell = \Theta(n)$. If required n can always be made larger by connecting dummy vertices to the root of T. (The resulting graph G_n is as in Figure 3 in Section 5.5.)* □

To prove the theorem, let $\mathcal{A}$ be any algorithm for the Path-Verification problem that solves the problem on G_n in at most k' rounds. We need some definitions and claims to prove the theorem.

Definitions of *left/right subtrees* and *breakpoints*. Consider a tree T' obtained by deleting all edges in P. Notice that nodes $v_{jk'+i}$, for all j and $i \leq k'/2$ are in the subtree of T' rooted at l and all remaining points are in the subtree rooted at r. For any node v, let $sub(v)$ denote the subtree rooted at node v. (Note that $sub(v)$ also

include nodes in the path P.) We denote the set of nodes that are leaves of $sub(l)$ by L (i.e., $L = sub(l) \cap P$) and the set of nodes that are leaves in $sub(r)$ by R.

Since we consider an algorithm that takes at most k rounds, consider the situation when the algorithm is given k rounds for *free* to communicate only along the edges of the path P at the beginning. Since L and R consists of every $k'/2$ vertices in P and $k'/2 > 2k$, there are some nodes unreachable from L by walking on P for k steps. In particular, all nodes of the form $v_{jk'+k'/2+k+1}$, for all j, are not reachable from L. We call such nodes *breakpoints* for $sub(l)$. Similarly all nodes of the form $v_{jk'+k+1}$, for all j, are not reachable from R and we call them the breakpoints for $sub(r)$. (See Figure 4 in Section 5.5.)

Definitions of *path-distance* and *covering*. For any two nodes u and v in T' (obtained from G_n by deleting edges in P), let $c(u, v)$ be a lowest common ancestor of u and v. We define $path\_dist(u, v)$ to be the number of leaves of subtree of T rooted at $c(u, v)$. Note that the path-distance is defined between any pair of nodes in G_n but the distance is counted using the number of leaves in T (which excludes nodes in P). (See Figure 5(a) in Section 5.5.)

We also introduce the notion of the path-distance *covered* by a message. For any message m, the path-distance covered by m is the maximum path-distance taken over all nodes that have held the message m. That is, if m covers some nodes $v'_1, v'_2, ..., v'_k$ then the path-distance covered by m is the number of leaves in the subtrees of T rooted by $v'_1, v'_2, ..., v'_k$. Note that some leaves may be in more than one subtrees and they will be counted only once. Our construction makes the right and left subtrees have a large number of break points, as in the following lemma. (Proof can be found in Section 4.6.2.1.)

Lemma 4.3.4. *The number of breakpoints for the left subtree and for the right subtree are at least $\frac{n}{4k}$ each.*

The reason we define these breakpoints is to show that the entire information held by the left subtree has many disjoint intervals, and same for the right subtree. This then tells us that the left subtree and the right subtree must *communicate* a lot to be able to merge these intervals by connecting/communicating the break points. To argue this, we show that the total path distance (over all messages) is large, as in the following lemma. (Proof is in Section 4.6.2.2.)

Lemma 4.3.5. *For algorithm $\mathcal{A}$ to solve* Path-Verification *problem, the total path-distance covered by all messages is at least n.*

These messages can however be communicated using the tree edges as well. We bound the maximum communication that can be achieved across $sub(l)$ and $sub(r)$ indirectly by bounding the maximum path-distance that can be covered in each round. In particular, we show the following lemma. See Figure 5(c) and proof in Section 5.5.

Lemma 4.3.6. *In k rounds, all messages together can cover at most a path-distance of $O(k^2 \log k)$.*

We now describe the proof of the main theorem using these three claims.

Proof of Theorem 4.3.2. Use Lemmas 4.3.5 and 4.3.6 we know that if $\mathcal{A}$ solves Path-Verification, then it needs to cover a *path_dist* of n, but in k rounds it can only cover a *path_dist* of $O(k^2 \log k)$. But this is $o(n)$ since $k = \sqrt{\frac{n}{\log n}}$, contradiction. □

4.3.2 Reduction to Random Walk Problem

We now discuss how the lower bound for the Path Verification problem implies the lower bound of the random walk problem. The main difference between Path-Verification problem and the random walk problem is that in the former we can specify which path to verify while the latter problem generates different path each time. We show that the "bad" instance (G_n and P) in the previous section can be

modified so that with high probability, the generated random walk is "hard" to verify. The theorems below are stated for ℓ length walk/path instead of n as above. As previously stated, if it is desired that ℓ be $o(n)$, it is always possible to add dummy nodes.

Theorem 4.3.7. *For any n, there exists a graph G_n of $\Theta(n)$ vertices and diameter $O(\log n)$, and $\ell = \Theta(n)$ such that, with high probability, a random walk of length ℓ needs $\Omega(\sqrt{\frac{\ell}{\log \ell}})$ rounds.*

Proof. Theorem 4.3.2 can be generalized to the case where the path P has infinite capacity, as follows.

Theorem 4.3.8. *For any n and $\ell = \Theta(n)$, there exists a graph G_n of $O(n)$ vertices and diameter $O(\log n)$, and a path P of length ℓ such that any algorithm that solves the* Path-Verification *problem on G_n and P requires more than $\Omega(\sqrt{\frac{\ell}{\log \ell}})$ rounds, even if edges in P have large capacity (i.e., one can send larger sized messages in one step).*

Proof. This is because the proof of Theorem 4.3.2 only uses the congestion of edges in the tree T (imposed above P) to argue about the number of rounds. □

Now, we modify G_n to G'_n as follows. Recall that the path P in G_n has vertices $v_1, v_2, ..., v_{n'}$. For each $i = 1, 2, ..., n'$, we define the weight of an edge (v_i, v_{i+1}) to be $(2n)^{2i}$ (note that weighted graphs are equivalent to unweighted multigraphs in our model). By having more weight, these edges have more capacity as well. However, increasing capacity does not affect the claim as shown above. Observe that, when the walk is at the node v_i, the probability of walk will take the edge (v_i, v_{i+1}) is at least $1 - \frac{1}{n^2}$. Therefore, P is the resulting random walk with probability at least $1 - 1/n$. When the random walk path is P, it takes at least $\sqrt{\frac{n}{\log n}}$ rounds to verify, by Theorem 4.3.8. This completes the proof. We remark that this construction requires exponential in n number of edges (multiedges). For the distributed computing model,

this only translates to a larger bandwidth. The length ℓ is still comparable to the number of nodes. □

4.4 Applications

In this section, we present two applications of our algorithm.

4.4.1 A Distributed Algorithm for Random Spanning Tree

We now present an algorithm for generating a random spanning tree (RST) of an unweighted undirected network in $\tilde{O}(\sqrt{m}D)$ rounds with high probability. The approach is to simulate Aldous and Broder's [5, 32] RST algorithm which is as follows. First, pick one arbitrary node as a root. Then, perform a random walk from the root node until all nodes are visited. For each non-root node, output the edge that is used for its first visit. (That is, for each non-root node v, if the first time v is visited is t then we output the edge (u, v) where u is the node visited at time $t-1$.) The output edges clearly form a spanning tree and this spanning tree is shown to come from a uniform distribution among all spanning trees of the graph [5, 32]. The expected time of this algorithm is the expected cover time of the graph which is shown to be $O(mD)$ (in the worst case, i.e., for any undirected, unweighted graph) by Aleniunas et al. [6].

This algorithm can be simulated on the distributed network by our random walk algorithm as follows. The algorithm can be viewed in phases. Initially, we pick a root node arbitrarily and set $\ell = n$. In each phase, we run $\log n$ (different) walks of length ℓ starting from the root node (this takes $\tilde{O}(\sqrt{\ell D})$ rounds using our distributed random walk algorithm). If none of the $O(\log n)$ different walks cover all nodes (this can be easily checked in $O(D)$ time), we double the value of ℓ and start a new phase, i.e., perform again $\log n$ walks of length ℓ. The algorithm continues until one walk of length ℓ covers all nodes. We then use such walk to construct a random spanning tree: As the result of this walk, each node knows its position(s) in the walk (cf.

Section 4.2.2), i.e., it has a list of steps in the walk that it is visited. Therefore, each non-root node can pick an edge that is used in its first visit by communicating to its neighbors. Thus at the end of the algorithm, each node can know which of its adjacent edges belong to the output tree. (An additional $O(n)$ rounds may be used to deliver the resulting tree to a particular node if needed.)

We now analyze the number of rounds in term of τ, the expected cover time of the input graph. The algorithm takes $O(\log \tau)$ phases before $2\tau \leq \ell \leq 4\tau$, and since one of $\log n$ random walks of length 2τ will cover the input graph with high probability, the algorithm will stop with $\ell \leq 4\tau$ with high probability. Since each phase takes $\tilde{O}(\sqrt{\ell D})$ rounds, the total number of rounds is $\tilde{O}(\sqrt{\tau D})$ with high probability. Since $\tau = \tilde{O}(mD)$, we have the following theorem.

Theorem 4.4.1. *The algorithm described above generates a uniform random spanning tree in $\tilde{O}(\sqrt{m}D)$ rounds with high probability.*

4.4.2 Decentralized Estimation of Mixing Time

We now present an algorithm to estimate the mixing time of a graph from a specified source. Throughout this section, we assume that the graph is connected and non-bipartite (the conditions under which mixing time is well-defined). The main idea in estimating the mixing time is, given a source node, to run many random walks of length ℓ using the approach described in the previous section, and use these to estimate the distribution induced by the ℓ-length random walk. We then compare the distribution at length ℓ, with the stationary distribution to determine if they are *close*, and if not, double ℓ and retry. For this approach, one issue that we need to address is how to compare two distributions with few samples efficiently (a well-studied problem). We introduce some definitions before formalizing our approach and theorem.

Definition 4.4.2 (Distribution vector)**.** *Let $\pi_x(t)$ define the probability distribution*

vector reached after t steps when the initial distribution starts with probability 1 at node x. Let π denote the stationary distribution vector.

Definition 4.4.3 ($\tau^x(\epsilon)$ and τ^x_{mix}, mixing time for source x)**.** *Define $\tau^x(\epsilon) = \min t : ||\pi_x(t) - \pi||_1 < \epsilon$. Define $\tau^x_{mix} = \tau^x(1/2e)$.*

The goal is to estimate τ^x_{mix}. Notice that the definition of τ^x_{mix} is consistent due to the following standard monotonicity property of distributions (proof in the Section 5.5).

Lemma 4.4.4. $||\pi_x(t+1) - \pi||_1 \leq ||\pi_x(t) - \pi||_1$.

To compare two distributions, we use the technique of Batu et. al. [21] to determine if the distributions are ϵ-near. Their result (slightly restated) is summarized in the following theorem.

Theorem 4.4.5 ([21])**.** *For any ϵ, given $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples of a distribution X over $[n]$, and a specified distribution Y, there is a test that outputs PASS with high probability if $|X - Y|_1 \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, and outputs FAIL with high probability if $|X - Y|_1 \geq 6\epsilon$.*

We now give a very brief description of the algorithm of Batu et. al. [21] to illustrate that it can in fact be simulated on the distributed network efficiently. The algorithm partitions the set of nodes in to buckets based on the steady state probabilities. Each of the $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from X now falls in one of these buckets. Further, the actual count of number of nodes in these buckets for distribution Y are counted. The exact count for Y for at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets (corresponding to the samples) is compared with the number of samples from X; these are compared to determine if X and Y are close. We refer the reader to their paper [21] for a precise description.

Our algorithm starts with $\ell = 1$ and runs $K = \tilde{O}(\sqrt{n})$ walks of length ℓ from the specified source x. As the test of comparison with the steady state distribution

outputs FAIL (for choice of $\epsilon = 1/12e$), ℓ is doubled. This process is repeated to identify the largest ℓ such that the test outputs FAIL with high probability and the smallest ℓ such that the test outputs PASS with high probability. These give lower and upper bounds on the required τ^x_{mix} respectively. Our resulting theorem is presented below and the proof is placed in Section 5.5.

Theorem 4.4.6. *Given a graph with diameter D, a node x can find, in $\tilde{O}(n^{1/2} + n^{1/4}\sqrt{D\tau^x(\epsilon)})$ rounds, a time $\tilde{\tau}^x_{mix}$ such that $\tau^x_{mix} \leq \tilde{\tau}^x_{mix} \leq \tau^x(\epsilon)$, where $\epsilon = \frac{1}{6912e\sqrt{n}\log n}$.*

Suppose our estimate of τ^x_{mix} is close to the mixing time of the graph defined as $\tau_{mix} = \max_x \tau^x_{mix}$, then this would allow us to estimate several related quantities. Given a mixing time τ_{mix}, we can approximate the spectral gap $(1 - \lambda_2)$ and the conductance (Φ) due to the known relations that $\frac{1}{1-\lambda_2} \leq \tau_{mix} \leq \frac{\log n}{1-\lambda_2}$ and $\Theta(1-\lambda_2) \leq \Phi \leq \Theta(\sqrt{1-\lambda_2})$ as shown in [90].

4.5 Discussion

This work makes progress towards resolving the time complexity of distributed computation of random walks in undirected networks. The dependence on the diameter D is still not tight, and it would be interesting to settle this. There is also a gap in our bounds for performing k independent random walks. Further, we look at the CONGEST model enforcing a bandwidth restriction and minimize number of rounds. While our algorithms have good *amortized* message complexity over several walks, it would be nice to come up with algorithms that are round efficient and yet have smaller message complexity.

We presented two algorithmic applications of our distributed random walk algorithm: estimating mixing times and computing random spanning trees. It would be interesting to improve upon these results. For example, is there a $\tilde{O}(\sqrt{\tau^x_{mix}} + n^{1/4})$ round algorithm to estimate τ^x; and is there a $\tilde{O}(n)$ round algorithm for RST?

There are several interesting directions to take this work further. Can these techniques be useful for estimating the second eigenvector of the transition matrix (useful for sparse cuts)? Are there efficient distributed algorithms for random walks in directed graphs (useful for PageRank and related quantities)? Finally, from a practical standpoint, it is important to develop algorithms that are robust to failures and it would be nice to extend our techniques to handle such node/edge failures.

4.6 Detailed Proofs and Figures

4.6.1 Omitted Proofs of Section 4.2 (Upper Bound)

4.6.1.1 Algorithm descriptions

The main algorithm for performing a single random walk is described in Single-Random-Walk (cf. Algorithm 13). This algorithm, in turn, uses Get-More-Walks (cf. 14 and Sample-Destination (cf. 15).

Notice that in Line 9 in Algorithm 14, the walks of length λ are extended further to walks of length $\lambda + r$ where r is a random number in the range $[0, \lambda - 1]$. We do this by extending the λ-length walks further, and probabilistically stopping each walk in each of the next i steps (for $0 \leq i \leq \lambda - 1$) with probability $\frac{1}{\lambda - i}$. The reason it needs to be done this way is because if we first sampled r, independently for each walk, in the range $[0, \lambda - 1]$ and then extended each walk accordingly, the algorithm would need to pass r independently for each walk. This will cause congestion along the edges; no congestion occurs in the mentioned algorithm as only the *count* of the number of walks along an edge are passed to the node across the edge.

4.6.1.2 Proofs of Lemma 4.2.1, 4.2.2, 4.2.3 and 4.2.4

Proof of Lemma 4.2.1. This proof is a slight modification of the proof of Lemma 2.2 in [51], where it is shown that each node can perform η walks of length λ together in $O(\lambda\eta \log n)$ rounds with high probability. We extend this to the following statement.

Algorithm 13 SINGLE-RANDOM-WALK(s, ℓ)

Input: Starting node s, and desired walk length ℓ.
Output: Destination node of the walk outputs the ID of s.

Phase 1: (Each node v performs $\eta_v = \eta \deg(v)$) random walks of length $\lambda + r_i$ where r_i (for each $1 \leq i \leq \eta$) is chosen independently at random in the range $[0, \lambda - 1]$.)

1: Let $r_{max} = \max_{1 \leq i \leq \eta} r_i$, the random numbers chosen independently for each of the η_x walks.
2: Each node x constructs η_x messages containing its ID and in addition, the i-th message contains the desired walk length of $\lambda + r_i$.
3: **for** $i = 1$ to $\lambda + r_{max}$ **do**
4: This is the i-th iteration. Each node v does the following: Consider each message M held by v and received in the $(i-1)$-th iteration (having current counter $i-1$). If the message M's desired walk length is at most i, then v stored the ID of the source (v is the desired destination). Else, v picks a neighbor u uniformly at random and forward M to u after incrementing its counter.
{Note that any iteration could require more than 1 round.}
5: **end for**

Phase 2: (Stitch $\Theta(\ell/\lambda)$ walks, each of length in $[\lambda, 2\lambda - 1]$)

1: The source node s creates a message called "token" which contains the ID of s
2: The algorithm generates a set of *connectors*, denoted by C, as follows.
3: Initialize $C = \{s\}$
4: **while** Length of walk completed is at most $\ell - 2\lambda$ **do**
5: Let v be the node that is currently holding the token.
6: v calls SAMPLE-DESTINATION(v) and let v' be the returned value (which is a destination of an unused random walk starting at v of length between λ and $2\lambda - 1$.)
7: **if** $v' =$ NULL (all walks from v have already been used up) **then**
8: v calls GET-MORE-WALKS(v, λ) (Perform $\Theta(l/\lambda)$ walks of length λ starting at v)
9: v calls SAMPLE-DESTINATION(v) and let v' be the returned value
10: **end if**
11: v sends the token to v'
12: $C = C \cup \{v\}$
13: **end while**
14: Walk naively until ℓ steps are completed (this is at most another 2λ steps)
15: A node holding the token outputs the ID of s

Algorithm 14 GET-MORE-WALKS(v, λ)

(Starting from node v, perform $\lfloor \ell/\lambda \rfloor$ number of random walks, each of length $\lambda + r_i$ where r_i is chosen uniformly at random in the range $[0, \lambda - 1]$ for the i-th walk.)

1: The node v constructs $\lfloor \ell/\lambda \rfloor$ (identical) messages containing its ID.
2: **for** $i = 1$ to λ **do**
3: Each node u does the following:
4: - For each message M held by u, pick a neighbor z uniformly at random as a receiver of M.
5: - For each neighbor z of u, send ID of v and the number of messages that z is picked as a receiver, denoted by $c(u, v)$.
6: - For each neighbor z of u, upon receiving ID of v and $c(u, v)$, constructs $c(u, v)$ messages, each contains the ID of v.
7: **end for**
{Each walk has now completed λ steps. These walks are now extended probabilistically further by r steps where each r is independent and uniform in the range $[0, \lambda - 1]$.}
8: **for** $i = 0$ to $\lambda - 1$ **do**
9: For each message, independently with probability $\frac{1}{\lambda - i}$, stop sending the message further and save the ID of the source node (in this event, the node with the message is the destination). For messages M that are not stopped, each node picks a neighbor correspondingly and sends the messages forward as before.
10: **end for**
11: At the end, each destination knows the source ID as well as the length of the corresponding walk.

Each node v can in fact perform $\eta \deg(v)$ of length 2λ and still finish in $O(\lambda \eta \log n)$ rounds.

The desired claim will follow immediately because each node v performs $\eta \deg(v)$ of length *at most* λ in Phase 1.

Consider the case when each node v creates $\eta \deg(v) \geq \eta$ messages. For each message M, any $j = 1, 2, ..., \lambda$, and any edge e, we define $X_M^j(e)$ to be a random variable having value 1 if M is sent through e in the j^{th} iteration (i.e., when the counter on M has value $j - 1$). Let $X^j(e) = \sum_{M:\text{message}} X_M^j(e)$. We compute the expected number of messages that go through an edge, see claim below.

Claim 4.6.1. *For any edge e and any j, $\mathbb{E}[X^j(e)] = 2\eta$.*

Proof. Assume that each node v starts with $\eta \deg(v)$ messages. Each message takes a random walk. We prove that after any given number of steps j, the expected number of messages at node v is still $\eta \deg(v)$. Consider the random walk's probability transition matrix, call it A. In this case $Au = u$ for the vector u having value $\frac{\deg(v)}{2m}$ where m is the number of edges in the graph (since this u is the stationary distribution of an undirected unweighted graph). Now the number of messages we started with at any node i is proportional to its stationary distribution, therefore, in expectation, the number of messages at any node remains the same.

To calculate $\mathbb{E}[X^j(e)]$, notice that edge e will receive messages from its two end points, say x and y. The number of messages it receives from node x in expectation is exactly the number of messages at x divided by $\deg(x)$. The claim follows. □

By Chernoff's bound (e.g., in [123, Theorem 4.4.]), for any edge e and any j,

$$\mathbb{P}[X^j(e) \geq 4\eta \log n] \leq 2^{-4 \log n} = n^{-4}.$$

It follows that the probability that there exists an edge e and an integer $1 \leq j \leq \lambda$ such that $X^j(e) \geq 4\eta \log n$ is at most $|E(G)|\lambda n^{-4} \leq \frac{1}{n}$ since $|E(G)| \leq n^2$ and $\lambda \leq \ell \leq n$ (by the way we define λ).

Now suppose that $X^j(e) \leq 4\eta \log n$ for every edge e and every integer $j \leq \lambda$. This implies that we can extend all walks of length i to length $i + 1$ in $4\eta \log n$ rounds. Therefore, we obtain walks of length λ in $4\lambda\eta \log n$ rounds as claimed. □

Proof of Lemma 4.2.2. The argument is exactly the same as the proof of Lemma 2.4 in [51]. That is, there is no congestion. We only consider longer walks (length at most $2\lambda - 1$) this time. The detail of the proof is as follows.

Consider any node v during the execution of the algorithm. If it contains x copies of the source ID, for some x, it has to pick x of its neighbors at random, and pass the source ID to each of these x neighbors. Although it might pass these messages to less than x neighbors, it sends only the source ID and a *count* to each neighbor, where the

count represents the number of copies of source ID it wishes to send to such neighbor. Note that there is only one source ID as one node calls GET-MORE-WALKS at a time. Therefore, there is no congestion and thus the algorithm terminates in $O(\lambda)$ rounds. □

Proof of Lemma 4.2.3. This proof is exactly the same as the proof of Lemma 2.5 in [51].

Constructing a BFS tree clearly takes only $O(D)$ rounds. In the second phase where the algorithm wishes to *sample* one of many tokens (having its ID) spread across the graph. The sampling is done while retracing the BFS tree starting from leaf nodes, eventually reaching the root. The main observation is that when a node receives multiple samples from its children, it only sends one of them to its parent. Therefore, there is no congestion. The total number of rounds required is therefore the number of levels in the BFS tree, $O(D)$. The third phase of the algorithm can be done by broadcasting (using a BFS tree) which needs $O(D)$ rounds. □

Proof of Lemma 4.2.4. The claim follows from the correctness of SAMPLE-DESTINATION that the algorithm samples a walk uniformly at random and the fact that the length of each walk is uniformly sampled from the range $[\lambda, 2\lambda - 1]$. The first part is proved in Lemma 2.6 in Das Sarma et al. [51] and included below for completeness. We now prove the second part.

To show that each walk length is uniformly sampled from the range $[\lambda, 2\lambda - 1]$, note that each walk can be created in two ways.

1. It is created in Phase 1. In this case, since we pick the length of each walk uniformly from the length $[\lambda, 2\lambda - 1]$, the claim clearly holds.

2. It is created by GET-MORE-WALK. In this case, the claim holds by the technique of *reservoir* sampling: Observe that after the λ^{th} step of the walk is completed, we stop extending each walk at any length between λ and $2\lambda - 1$

uniformly. To see this, observe that we stop at length λ with probability $1/\lambda$. If the walk does not stop, it will stop at length $\lambda+1$ with probability $\frac{1}{\lambda-1}$. This means that the walk will stop at length $\lambda+1$ with probability $\frac{\lambda-1}{\lambda} \times \frac{1}{\lambda-1} = \frac{1}{\lambda-1}$. Similarly, it can be argue that the walk will stop at length i for any $i \in [\lambda, 2\lambda-1]$ with probability $\frac{1}{\lambda}$.

We now show the proof of Lemma 2.6 (with slight modification) in Das Sarma et al. for completeness.

Lemma 4.6.2 (Lemma 2.6 in [51])**.** *Algorithm* Sample-Destination*(v) (cf. Algorithm 15), for any node v, samples a destination of a walk starting at v uniformly at random.*

Proof. Assume that before this algorithm starts, there are t (without loss of generality, let $t > 0$) "tokens" containing ID of v stored in some nodes in the network. The goal is to show that Sample-Destination brings one of these tokens to v with uniform probability. For any node u, let T_u be the subtree rooted at u and let S_u be the set of tokens in T_u. (Therefore, $T_v = T$ and $|S_v| = t$.)

We claim that any node u returns a destination to its parent with uniform probability (i.e., for any tokens $x \in S_u$, $Pr[u \text{ returns } x]$ is $1/|S_u|$ (if $|S_u| > 0$)). We prove this by induction on the height of the tree. This claim clearly holds for the base case where u is a leaf node. Now, for any non-leaf node u, assume that the claim is true for any of its children. To be precise, suppose that u receives tokens and counts from q children. Assume that it receives tokens $d_1, d_2, ..., d_q$ and counts $c_1, c_2, ..., c_q$ from nodes $u_1, u_2, ..., u_q$, respectively. (Also recall that d_0 is the sample of its own tokens (if exists) and c_0 is the number of its own tokens.) By induction, d_j is sent from u_j to u with probability $1/|S_{u_j}|$, for any $1 \leq j \leq q$. Moreover, $c_j = |S_{u_j}|$ for any j. Therefore, any token d_j will be picked with probability $\frac{1}{|S_{u_j}|} \times \frac{c_j}{c_0+c_1+...c_q} = \frac{1}{S_u}$ as claimed.

The lemma follows by applying the claim above to v. □

□

4.6.1.3 Proof of Lemma 4.2.7

Proof. Intuitively, this argument is simple, since the connectors are spread out in steps of length approximately λ. However, there might be some *periodicity* that results in the same node being visited multiple times but *exactly* at λ-intervals. This is where we crucially use the fact that the algorithm uses walks of length $\lambda + r$ where r is chosen uniformly at random from $[0, \lambda - 1]$.

We prove the lemma using the following two claims.

Claim 4.6.3. *Consider any sequence A of numbers $a_1, ..., a'_\ell$ of length ℓ'. For any integer λ', let B be a sequence $a_{\lambda'+r_1}, a_{2\lambda'+r_1+r_2}, ..., a_{i\lambda'+r_1+...+r_i}, ...$ where r_i, for any i, is a random integer picked uniformly from $[0, \lambda' - 1]$. Consider another subsequence of numbers C of A where an element in C is picked from from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$. Then, $Pr[C$ contains $a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}]$ for any set $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$.*

Proof. First consider a subsequence C of A. Numbers in C are picked from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$. Observe that $|C| \geq |B|$. In fact, we can say that "C contains B"; i.e., for any sequence of k indexes $i_1, i_2, ..., i_k$ such that $\lambda' \leq i_{j+1} - i_j \leq 2\lambda' - 1$ for all j,

$$Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = Pr[C \text{ contains } \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}].$$

To see this, observe that B will be equal to $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$ only for a specific value of $r_1, r_2, ..., r_k$. Since each of $r_1, r_2, ..., r_k$ is chosen uniformly at random from $[1, \lambda']$, $Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = \lambda'^{-k}$. Moreover, the C will contain $a_{i_1}, ai_2, ..., a_{i_k}\}$ if

and only if, for each j, we pick a_{i_j} from the interval that contains it (i.e., from $a_{(i'-1)\lambda'+1}, a_{(i'-1)\lambda'+2}, ..., a_{i'\lambda'}$, for some i'). (Note that $a_{i_1}, a_{i_2}, ...$ are all in different intervals because $i_{j+1} - i_j \geq \lambda'$ for all j.) Therefore, $Pr[C \text{ contains } a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = \lambda'^{-k}$. □

Claim 4.6.4. *Consider any sequence A of numbers $a_1, ..., a'_\ell$ of length ℓ'. Consider subsequence of numbers C of A where an element in C is picked from from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$.. For any number x, let n_x be the number of appearances of x in A; i.e., $n_x = |\{i \mid a_i = x\}|$. Then, for any $R \geq 6n_x/\lambda'$, x appears in C more than R times with probability at most 2^{-R}.*

Proof. For $i = 1, 2, ..., \lfloor \ell'/\lambda' \rfloor$, let X_i be a 0/1 random variable that is 1 if and only if $c_i = x$ and $X = \sum_{i=1}^{\lfloor \ell'/\lambda' \rfloor} X_i$. That is, X is the number of appearances of x in C. Clearly, $E[X] = n_x/\lambda'$. Since X_i's are independent, we can apply the Chernoff bound (e.g., in [123, Theorem 4.4.]): For any $R \geq 6E[X] = 6n_x/\lambda'$,

$$Pr[X \leq R] \geq 2^{-R}.$$

The claim is thus proved. □

Now we use the claim to prove the lemma. Choose $\ell' = \ell$ and $\lambda' = \lambda$ and consider any node v that appears at most t times. The number of times it appears as a connector node is the number of times it appears in the subsequence B described in the claim. By applying the claim with $R = t(\log n)^2$, we have that v appears in B more than $t(\log n)^2$ times with probability at most $1/n^2$ as desired. □

4.6.1.4 Proof of Lemma 4.2.6

We start with the bound of the first and second moment of the number of visits at each node by each walk.

Proposition 4.6.5. *For any node x, node y and $t = O(m^2)$,*

$$\mathrm{E}[N_t^x(y)] \leq 8d(y)\sqrt{t+1}\,, \quad \text{and} \quad \mathrm{E}\Big[\big(N_t^x(y)\big)^2\Big] \leq \mathrm{E}[N_t^x(y)] + 128\; d^2(y)\;(t+1)\,. \tag{1}$$

To prove the above proposition, let P denote the transition probability matrix of such a random walk and let π denote the stationary distribution of the walk, which in this case is simply proportional to the degree of the vertex.

The basic bound we use is the following estimate from Lyons (see Lemma 3.4 and Remark 4 in [116]). Let Q denote the transition probability matrix of a chain with self-loop probablity $\alpha > 0$, and with $c = \min\{\pi(x)Q(x,y) : x \neq y \text{ and } Q(x,y) > 0\}$. Note that for a random walk on an undirected graph, $c = \frac{1}{2m}$. For $k > 0$ a positive integer (denoting time) ,

$$\Big|\frac{Q^k(x,y)}{\pi(y)} - 1\Big| \leq \min\Big\{\frac{1}{\alpha c\sqrt{k+1}}, \frac{1}{2\alpha^2 c^2(k+1)}\Big\}\,. \tag{2}$$

For $k \leq \beta m^2$ for a sufficiently small constant β, and small α, the above can be simplified to the following bound; see Remark 3 in [116].

$$Q^k(x,y) \leq \frac{4\pi(y)}{c\sqrt{k+1}} = \frac{4d(y)}{\sqrt{k+1}}\,. \tag{3}$$

Note that given a simple random walk on a graph G, and a corresponding matrix P, one can always switch to the lazy version $Q = (I+P)/2$, and interpret it as a walk on graph G', obtained by adding self-loops to vertices in G so as to double the degree of each vertex. In the following, with abuse of notation we assume our P is such a lazy version of the original one.

Proof. Let $X_0, X_1, \ldots$ describe the random walk, with X_i denoting the position of the walk at time $i \geq 0$, and let $\mathbf{1}_A$ denote the indicator (0-1) random variable, which takes the value 1 when the event A is true. In the following we also use the subscript x to denote the fact that the probability or expectation is with respect to starting

the walk at vertex x. First the expectation.

$$\begin{aligned}
\mathrm{E}[N_t^x(y)] &= \mathrm{E}_x[\sum_{i=0}^{t} \mathbf{1}_{\{X_i=y\}}] = \sum_{i=0}^{t} P^i(x,y) \\
&\leq 4d(y) \sum_{i=0}^{t} \frac{1}{\sqrt{i+1}}, \quad \text{(using the above inequality (3))} \\
&\leq 8d(y)\sqrt{t+1}\,.
\end{aligned}$$

Abbreviating $N_t^x(y)$ as $N_t(y)$, we now compute the second moment:

$$\begin{aligned}
\mathrm{E}[N_t^2(y)] &= \mathrm{E}_x\Big[\big(\sum_{i=0}^{t} \mathbf{1}_{\{X_i=y\}}\big)\big(\sum_{j=0}^{t} \mathbf{1}_{\{X_j=y\}}\big)\Big] \\
&= \mathrm{E}_x\Big[\sum_{i=0}^{t} \mathbf{1}_{\{X_i=y\}} + 2 \sum_{0\leq i<j\leq t} \mathbf{1}_{\{X_i=y,\ X_j=y\}}\Big] \\
&= \mathrm{E}[N_t(y)] + 2 \sum_{0\leq i<j\leq t} \Pr(X_i = y,\ X_j = y)\,.
\end{aligned}$$

To bound the second term on the right hand side above, consider for $0 \leq i < j$:

$$\begin{aligned}
\Pr(X_i = y,\ X_j = y) &= \Pr(X_i = y)\ \Pr(X_j = y | X_i = y) \\
&= P^i(x,y)\ \ P^{j-i}(y,y)\,, \quad \text{due to the Markovian property} \\
&\leq \frac{4d(y)}{\sqrt{i+1}}\ \ \frac{4d(y)}{\sqrt{j-i+1}}\,. \quad \text{(using (3))}
\end{aligned}$$

Thus,

$$\begin{aligned}
\sum_{0\leq i<j\leq t} \Pr(X_i = y,\ X_j = y) &\leq \sum_{0\leq i\leq t} \frac{4d(y)}{\sqrt{i+1}} \sum_{0<j-i\leq t-i} \frac{4d(y)}{\sqrt{j-i+1}} \\
&= 16d^2(y) \sum_{0\leq i\leq t} \frac{1}{\sqrt{i+1}} \sum_{0<k\leq t-i} \frac{1}{\sqrt{k+1}} \\
&\leq 32d^2(y) \sum_{0\leq i\leq t} \frac{1}{\sqrt{i+1}}\ \sqrt{t-i+1} \\
&\leq 32d^2(y)\sqrt{t+1} \sum_{0\leq i\leq t} \frac{1}{\sqrt{i+1}} \\
&\leq 64d^2(y)\ (t+1)\,,
\end{aligned}$$

which yields the claimed bound on the second moment in the proposition. □

Using the above proposition, we bound the number of visits of each walk at each node, as follows.

Lemma 4.6.6. *For $t = O(m^2)$ and any vertex $y \in G$, the random walk started at x satisfies:*

$$\Pr\big(N_t^x(y) \geq 24\ d(y)\sqrt{t+1}\log n\big) \leq \frac{1}{n^2}\,.$$

Proof. First, it follows from the Proposition that

$$\Pr\big(N_t^x(y) \geq 2 \cdot 12\ d(y)\sqrt{t+1}\big) \leq \frac{1}{4}\,. \tag{4}$$

This is done by using the standard Chebyshev argument that for $B > 0$, $\Pr\big(N_t(y) \geq B\big) \leq \Pr\big(N_t^2(y) \geq B^2\big) \leq \frac{\mathrm{E}\left(N_t^2(y)\right)}{B^2}$.

For any r, let $L_r^x(y)$ be the time that the random walk (started at x) visits y for the r^{th} time. Observe that, for any r, $N_t^x(y) \geq r$ if and only if $L_r^x(y) \leq t$. Therefore,

$$\Pr(N_t^x(y) \geq r) = \Pr(L_r^x(y) \leq t). \tag{5}$$

Let $r^* = 24\ d(y)\sqrt{t+1}$. By (4) and (5), $\Pr(L_{r^*}^x(y) \leq t) \leq \frac{1}{4}$. We claim that

$$\Pr(L_{r^*\log n}^x(y) \leq t) \leq \left(\frac{1}{4}\right)^{\log n} = \frac{1}{n^2}\,. \tag{6}$$

To see this, divide the walk into $\log n$ independent subwalks, each visiting y exactly r^* times. Since the event $L_{r^*\log n}^x(y) \leq t$ implies that all subwalks have length at most t, (6) follows. Now, by applying (5) again,

$$\Pr(N_t^x(y) \geq r^* \log n) = \Pr(L_{r^*\log n}^x(y) \leq t) \leq \frac{1}{n^2}$$

as desired.

□

We now extend the above lemma to bound the number of visits of *all* the walks at each particular node.

Lemma 4.6.7. *For $\gamma > 0$, and $t = O(m^2)$, and for any vertex $y \in G$, the random walk started at x satisfies:*

$$\Pr\Big(\sum_{i=1}^{k} N_t^{x_i}(y) \geq 24\ d(y)\sqrt{kt+1}\log n + k\Big) \leq \frac{1}{n^2}\,.$$

Proof. First, observe that, for any r,

$$\Pr\Big(\sum_{i=1}^{k} N_t^{x_i}(y) \geq r - k\Big) \leq \Pr[N_{kt}^{y}(y) \geq r].$$

To see this, we construct a walk W of length kt starting at y in the following way: For each i, denote a walk of length t starting at x_i by W_i. Let τ_i and τ_i' be the first and last time (not later than time t) that W_i visits y. Let W_i' be the subwalk of W_i from time τ_i to τ_i'. We construct a walk W by stitching $W_1', W_2', ..., W_k'$ together and complete the rest of the walk (to reach the length kt) by a normal random walk. It then follows that the number of visits to y by $W_1, W_2, \ldots, W_k$ (excluding the starting step) is at most the number of visits to y by W. The first quantity is $\sum_{i=1}^{k} N_t^{x_i}(y) - k$. (The term '$-k$' comes from the fact that we do not count the first visit to y by each W_i which is the starting step of each W_i'.) The second quantity is $N_{kt}^{y}(y)$. The observation thus follows.

Therefore,

$$\Pr\Big(\sum_{i=1}^{k} N_t^{x_i}(y) \geq 24\, d(y)\sqrt{kt+1}\log n + k\Big) \leq \Pr\Big(N_{kt}^{y}(y) \geq 24\, d(y)\sqrt{kt+1}\log n\Big) \leq \frac{1}{n^2}$$

where the last inequality follows from Lemma 4.6.6. □

Lemma 4.2.6 follows immediately from Lemma 4.6.7 by union bounding over all nodes.

4.6.1.5 Proof of Theorem 4.2.8

Proof. First, consider the case where $\lambda > \ell$. In this case, $\min(\sqrt{k\ell D} + k, \sqrt{k\ell} + k + \ell) = \tilde{O}(\sqrt{k\ell} + k + \ell)$. By Lemma 4.2.6, each node x will be visited at most

$\tilde{O}(d(x)(\sqrt{k\ell}+k))$ times. Therefore, using the same argument as Lemma 4.2.1, the congestion is $\tilde{O}(\sqrt{k\ell}+k)$ with high probability. Since the dilation is ℓ, Many-Random-Walks takes $\tilde{O}(\sqrt{k\ell}+k+\ell)$ rounds as claimed. Since $2\sqrt{k\ell} \leq k+\ell$, this bound reduces to $O(k+\ell)$.

Now, consider the other case where $\lambda \leq \ell$. In this case, $\min(\sqrt{k\ell D}+k, \sqrt{k\ell}+k+\ell) = \tilde{O}(\sqrt{k\ell D}+k)$. Phase 1 takes $\tilde{O}(\lambda\eta) = \tilde{O}(\sqrt{k\ell D}+k)$. The stitching in Phase 2 takes $\tilde{O}(k\ell D/\lambda) = \tilde{O}(\sqrt{k\ell D})$. Moreover, by Lemma 4.2.6, Get-More-Walks will never be invoked. Therefore, the total number of rounds is $\tilde{O}(\sqrt{k\ell D}+k)$ as claimed. □

4.6.2 Omitted Proofs of Section 4.3 (Lower Bound)

4.6.2.1 Proof of Lemma 4.3.4

Proof. After the first k free rounds, consider the intervals that the left subtree can have, in the best case. Recall that these k rounds allowed communication only along the path. The *path_dist* of any node in L from the breakpoints of $sub(L)$ along the path is at least $k+1$. □

4.6.2.2 Proof of Lemma 4.3.5

Proof. First, notice that each left breakpoint is at a path-distance of $k+1$ from every node in the right subtree. That is, $path\_dist(u, L) = path\_dist(v, R) = k+1$ for all $u \in B_l$ and all $v \in B_r$.

Each breakpoint needs to be combined into one interval in the end. However, there could be one interval that is communicated from the $sub(l)$ to the $sub(r)$ (or vice versa) such that it connects several breakpoints. We show that this cannot happen. Consider all the breakpoints $v \in B_l \cup B_r$.

Definition of *scratching*.

Let us say that we *scratch out* the breakpoints from the list $k+1$, $k'/2+k+1$, $k'+k+1$, $k'+k'/2+k+1$, $2k'+k+1$, ... that get connected when an interval is

communicated between $sub(l)$ and $sub(r)$. We scratch out a breakpoint if there is an interval in the graph that contains it and both (or one in case of the first and last breakpoints) its adjacent breakpoints. For example, if the left subtree has intervals $[1, k'/2+k]$ and $[k'/2+k+2, k'+k'/2+k+1]$ and the right subtree has $[k+2, k'+k]$ and the latter interval is communicated to a node in the left subtree, then the left subtree is able to obtain the merged interval $[1, k' + k'/2 + k + 1]$ and therefore breakpoints $k + 1$ and $k'/2 + k + 1$ are scratched out.

Claim 4.6.8. *At most $O(1)$ breakpoints can be scratched out with one message/interval communicated between $sub(r)$ and $sub(l)$*

Proof. We argue that with the communication of one interval across the left and right subtrees, at most 4 breakpoints that have not been scratched yet can get scratched. This follows from a simple inductive argument. Consider a situation where the left subtree has certain intervals with all overlapping intervals already merged, and similarly right subtree. Suppose an interval $\mathcal{I}$ is communicated between $sub(r)$ and $sub(l)$, one of the following cases arise:

- $\mathcal{I}$ contains one breakpoint: Can be merged with at most two other intervals. Therefore, at most three breakpoints can get scratched.
- $\mathcal{I}$ contains two breakpoints: Can get connected with at most two other intervals and therefore at most four breakpoints can get scratched.
- $\mathcal{I}$ contains more than two breakpoints: This is impossible since there are at most two breakpoints in each interval, its left most and right most numbers (by definition of scratching).

This completes the proof of the claim. □

The proof now follows from Lemma 4.3.4. For any breakpoint b, let M_b be the set of messages that represents an interval containing b while b is still unscratched. If b is in $sub(l)$ and gets scratched because of the combination of some intervals in

$sub(r)$, then we claim that M_b has covered a path-distance of at least k. (Define the path-distance covered by M_b by the total path-distance covered by all messages in M_b.) This is because $b = v_i$ (say), being a breakpoint in $sub(l)$ has i equal to $(k+1 \mod k')$. Therefore, b is at a path distance of at least k from any node in R. Consequently, b is at a path-distance of at least k from any node in $sub(r)$. Since there are $\Theta(\frac{n}{4k})$ breakpoints, and for any interval to be communicated across the left and right subtree, a path-distance of k must be covered, in total, $\Theta(n)$ path-distance must be covered for all breakpoints to be scratched. This follows from three main observations:

- As shown above, for any breakpoint to be scratched, an interval with a breakpoint must be communicated from $sub(l)$ to $sub(r)$ or vice versa (thereby all messages m containing the breakpoint together covering a path-distance of at least k)
- Any message/interval with unscratched breakpoints has at most two unscratched breakpoints
- As shown in Claim 4.6.8, at most four breakpoints can be scratched when two intervals are merged.

  The proof follows. (Also see Figure 5(b) for the idea of this proof.) □

4.6.2.3 Proof of Lemma 4.3.6

Proof. We consider the total number of messages that can go through nodes at any level of the graph, starting from level 0 to level $\log k$ under the congest model.

First notice that if a message is passed at level i of the tree, this can cover a *path_dist* of at most 2^i. This is because the subtree rooted at a node at level i has 2^i leaves. Further, by our construction, there are $2^{\log(k')-i}$ nodes at level i. Therefore, all nodes at level i together, in a given round of $\mathcal{A}$ can cover a $dist - path$, path distance, of at most $2^i 2^{\log(k')-i} = 4k+2$. Therefore, over k rounds, the total *path_dist* that can be covered in a single level is $k(k')$. Since there are $O(\log k)$ levels, the total

path_dist that can be covered in k rounds over the entire graph is $O(k^2 \log k)$. (See Figure 5(c).) □

4.6.3 Omitted Proofs of Section 4.4.2 (Mixing Time)

4.6.3.1 Brief description of algorithm for Theorem 4.4.5

The algorithm partitions the set of nodes in to buckets based on the steady state probabilities. Each of the $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from X now falls in one of these buckets. Further, the actual count of number of nodes in these buckets for distribution Y are counted. The exact count for Y for at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets (corresponding to the samples) is compared with the number of samples from X; these are compared to determine if X and Y are close. We refer the reader to their paper [21] for a precise description.

4.6.3.2 Proof of Lemma 4.4.4

Proof. The monotonicity follows from the fact that $||Ax||_1 \leq ||x||_1$ where A is the transpose of the transition probability matrix of the graph and x is any probability vector. That is, $A(i, j)$ denotes the probability of transitioning from node j to node i. This in turn follows from the fact that the sum of entries of any column of A is 1.

Now let π be the stationary distribution of the transition matrix A. This implies that if ℓ is ϵ-near mixing, then $||A^l u - \pi||_1 \leq \epsilon$, by definition of ϵ-near mixing time. Now consider $||A^{l+1}u - \pi||_1$. This is equal to $||A^{l+1}u - A\pi||_1$ since $A\pi = \pi$. However, this reduces to $||A(A^l u - \pi)||_1 \leq \epsilon$. It follows that $(\ell + 1)$ is ϵ-near mixing. □

4.6.3.3 Proof of Theorem 4.4.6

Proof. For undirected unweighted graphs, the stationary distribution of the random walk is known and is $\frac{deg(i)}{2m}$ for node i with degree $deg(i)$, where m is the number of edges in the graph. If a source node in the network knows the degree distribution, we only need $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from a distribution to compare it to the stationary

distribution. This can be achieved by running MultipleRandomWalk to obtain $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ random walks. We choose $\epsilon = 1/12e$. To find the approximate mixing time, we try out increasing values of l that are powers of 2. Once we find the right consecutive powers of 2, the monotonicity property admits a binary search to determine the exact value for the specified ϵ.

The result in [21] can also be adapted to compare with the steady state distribution even if the source does not know the entire distribution. As described previously, the source only needs to know the *count* of number of nodes with steady state distribution in given buckets. Specifically, the buckets of interest are at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ as the count is required only for buckets were a sample is drawn from. Since each node knows its own steady state probability (determined just by its degree), the source can broadcast a specific bucket information and recover, in $O(D)$ steps, the count of number of nodes that fall into this bucket. Using the standard upcast technique previously described, the source can obtain the bucket count for each of these at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets in $\tilde{O}(n^{1/2}poly(\epsilon^{-1}) + D)$ rounds.

We have shown previously that a source node can obtain K samples from K independent random walks of length ℓ in $\tilde{O}(K + \sqrt{KlD})$ rounds. Setting $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}) + D)$ completes the proof. □

4.6.4 Figures

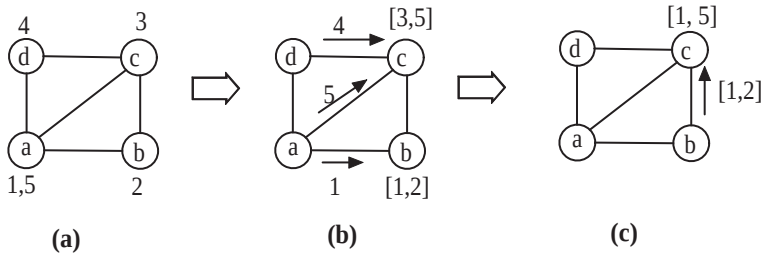


Figure 1: Example of path verification problem. **(a)** In the beginning, we want to verify that the vertices containing numbers 1..5 form a path. (In this case, they form a path a, b, c, d, a.) **(b)** One way to do this is for a to send 1 to b and therefore b can check that two vertices a and b corresponds to label 1 and 2 form a path. (The interval $[1, 2]$ is used to represent the fact that vertices corresponding to numbers $1, 2$ are verified to form a path.) Similarly, c can verify $[3, 5]$. **(c)** Finally, c combine $[1, 2]$ with $[3, 5]$ and thus the path corresponds to numbers $1, 2, ..., 5$ is verified.

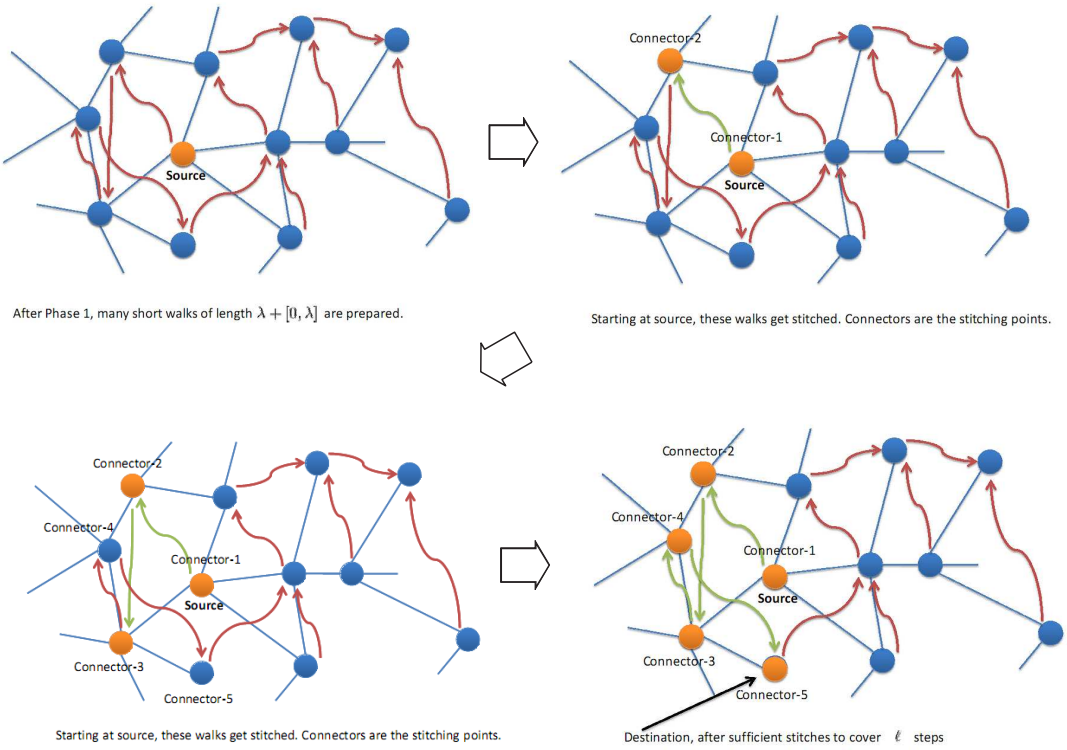


Figure 2: Figure illustrating the Algorithm of stitching short walks together.

Algorithm 15 SAMPLE-DESTINATION(v)

Input: Starting node v.

Output: A node sampled from among the stored walks (of length in $[\lambda, 2\lambda - 1]$) from v.

Sweep 1: (Perform BFS tree)

1: Construct a Breadth-First-Search (BFS) tree rooted at v. While constructing, every node stores its parent's ID. Denote such tree by T.

Sweep 2: (Tokens travel up the tree, sample as you go)

1: We divide T naturally into levels 0 through D (where nodes in level D are leaf nodes and the root node s is in level 0).
2: Tokens are held by nodes as a result of doing walks of length between λ and $2\lambda-1$ from v (which is done in either Phase 1 or GET-MORE-WALKS (cf. Algorithm 14)) A node could have more than one token.
3: Every node u that holds token(s) picks one token, denoted by d_0, uniformly at random and lets c_0 denote the number of tokens it has.
4: **for** $i = D$ down to 0 **do**
5: Every node u in level i that either receives token(s) from children or possesses token(s) itself do the following.
6: Let u have tokens $d_0, d_1, d_2, \ldots, d_q$, with counts $c_0, c_1, c_2, \ldots, c_q$ (including its own tokens). The node v samples one of d_0 through d_q, with probabilities proportional to the respective counts. That is, for any $1 \leq j \leq q$, d_j is sampled with probability $\frac{c_j}{c_0+c_1+\ldots+c_q}$.
7: The sampled token is sent to the parent node (unless already at root), along with a count of $c_0 + c_1 + \ldots + c_q$ (the count represents the number of tokens from which this token has been sampled).
8: **end for**
9: The root output the ID of the owner of the final sampled token. Denote such node by u_d.

Sweep 3: (Go and delete the sampled destination)

1: v sends a message to u_d (e.g., via broadcasting). u_d deletes one token of v it is holding (so that this random walk of length λ is not reused/re-stitched).

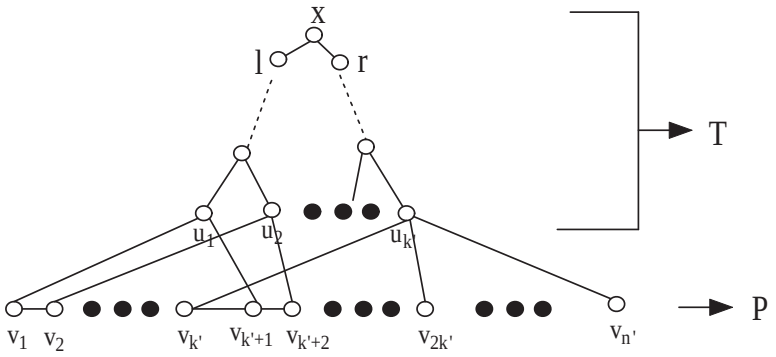


Figure 3: G_n

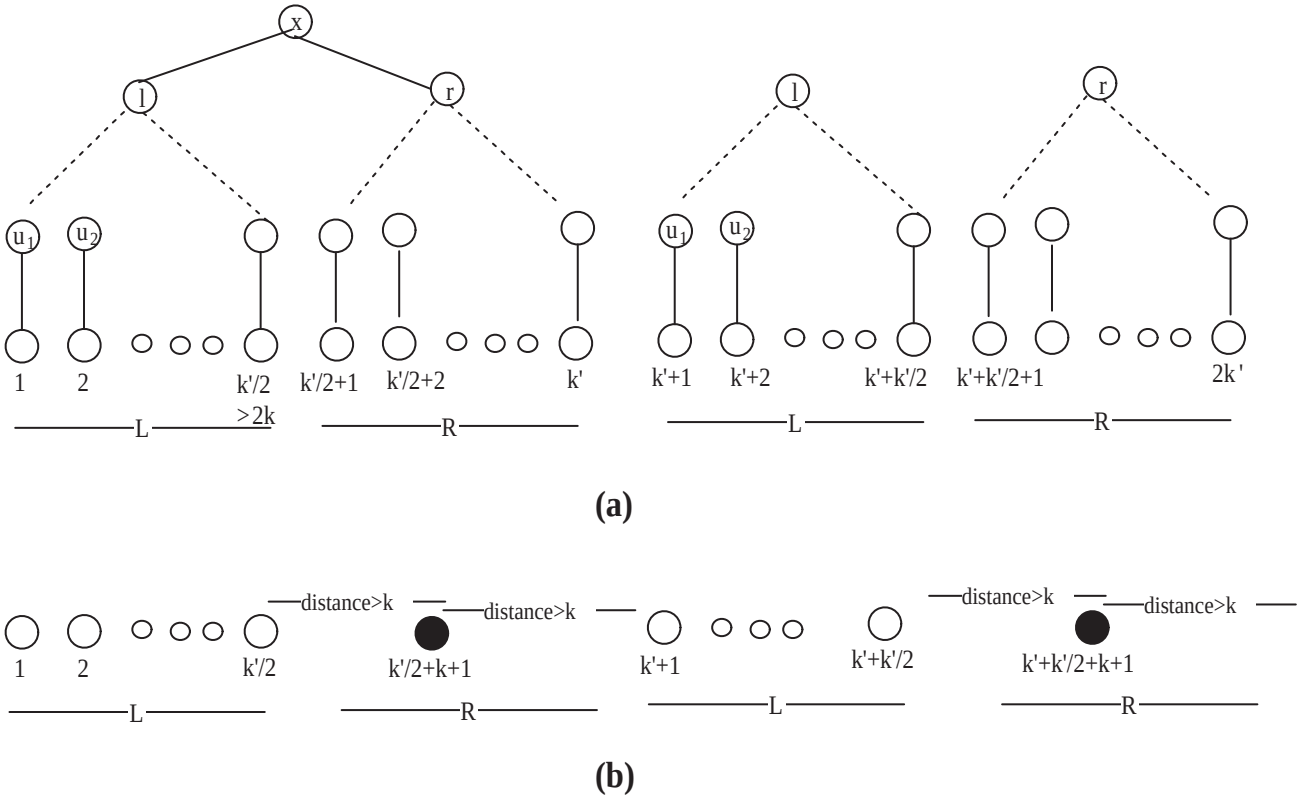
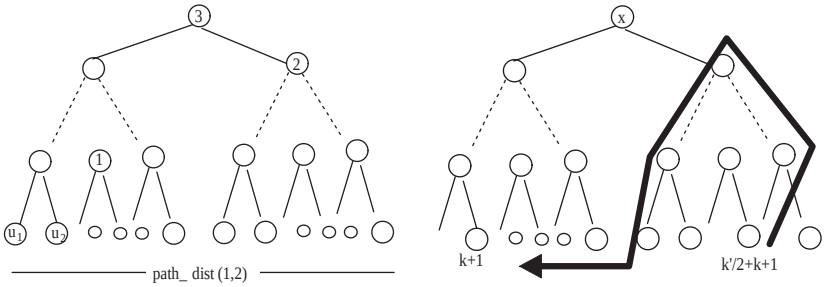
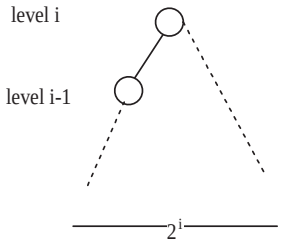


Figure 4: Breakpoints. **(a)** L and R consist of every other $k'/2$ vertices in P. (Note that we show the vertices l and r appear many times for the convenience of presentation.) **(b)** $v_{k'/2+k+1}$ and $v_{k'+k'/2+k+1}$ (nodes in black) are two of the breakpoints for L. Notice that there is one breakpoint in every connected piece of L and R.



(a) Path-distance. (b) Idea of Claim 4.6.8



(c) Idea of Lemma 4.3.6.

Figure 5: (a) Path distance between 1 and 2 is the number of leaves in the subtree rooted at 3, the lowest common ancestor of 1 and 2. **(b)** For one unscratched left breakpoint, $k'/2 + k + 1$ to be combined with another right breakpoint $k + 1$ on the left, $k'/2 + k + 1$ has to be carried to L by some intervals. Moreover, one interval can carry at most two unscratched breakpoints at a time. **(c)** Sending a message between nodes on level i and $i - 1$ can increase the covered path distance by at most 2^i.

CHAPTER V

SPARSE CUT PROJECTIONS IN GRAPH STREAMS

Finding sparse cuts is an important tool for analyzing large graphs that arise in practice, such as the web graph, online social communities, and VLSI circuits. When dealing with such graphs having billions of nodes, it is often hard to visualize global partitions. While studies on sparse cuts have traditionally looked at cuts with respect to all the nodes in the graph, some recent works analyze graph properties projected onto a small subset of vertices that may be of interest in a given context, e.g., relevant documents to a query in a search engine. In this chapter, we study how sparse cuts in a graph partition a certain subset of nodes. We call this partition a *cut projection*. We [49] study the problem of finding cut projections in the streaming model that is appropriate in this context as the input graph is too large to store in main memory. Specifically, for a d-regular graph G on n nodes with a cut of conductance Φ and constant balance, we show how to partition a randomly chosen set of k nodes in $\tilde{O}(\frac{1}{\sqrt{\alpha\Phi}})$ passes over the graph stream and space $\tilde{O}(n\alpha + \frac{n^{3/4}k^{1/4}}{\sqrt{\alpha}\Phi^{19/4}})$, for any choice of $\alpha \leq 1$. The resulting partition is the projection of a cut of conductance of at most $\tilde{O}(\sqrt{\Phi})$. We note that for $k < n\alpha^6\Phi^{O(1)}$, this can be done in $\tilde{O}(1/\sqrt{\alpha\Phi})$ passes and space $\tilde{O}(n\alpha)$ that is sublinear in the number of nodes.

5.1 Related Work and Contributions

The problem of finding sparse cuts on graphs has been studied extensively [25, 22, 92, 13, 136, 118]. Sparse cuts form an important tool for analyzing/partitioning real world graphs, such as the web graph, click graphs from search engine query logs and online social communities [29]. While traditionally studies on sparse cuts have looked at cuts with respect to all the nodes in the graph, more recent works [108, 107] study

graph properties projected onto a small subset of nodes that may be of interest. A similar approach of understanding the structure of the graph by looking at smaller sets of nodes has been taken in several studies including HITS [100], SALSA [106], and web projections [107], where they study the properties of the web graph restricted to a set of documents that match a given query.

In this work, we present a streaming algorithm for finding how a sparse cut partitions a small random set of nodes when the graph is presented as a stream of edges in no particular order. Our streaming algorithm uses space sublinear in the number of nodes. We also provide an algorithm for finding a sparse cut on the entire graph. We now introduce some definitions below.

Definition 5.1.1 (**Conductance and Sparsity**). *The conductance of a graph* $G = (V, E)$ *of* n *nodes* $1, 2, \ldots, n$ *is defined as* $\Phi(G) = \min_{S:E(S)\leq E(V)/2} \frac{E(S,V\setminus S)}{E(S)}$ *where* $E(S, V \setminus S)$ *is the number of edges crossing the cut* $(S, V \setminus S)$ *and* $E(S)$ *is the number of edges with at least one end point incident on* S. *For* d*-regular graphs,* $\Phi(G) = \min_{S:|S|\leq|V|/2} \frac{E(S,V\setminus S)}{d|S|}$. *Further, this is within a factor two of* $\min_S \frac{nE(S,V\setminus S)}{d|S||V\setminus S|}$. *We also note that the sparsity of a* d*-regular graph is related to the conductance by a factor* d.

Definition 5.1.2 (**Balance**). *The balance of a cut* $(S, V\setminus S)$ *is defined as* $\min\{\frac{|V\setminus S|}{|V|}, \frac{|S|}{|V|}\}$.

Definition 5.1.3 (**Cut Projections**). *Given a cut* $(S, V \setminus S)$, *we will say that* $(S \cap U, V \setminus S \cap U)$ *is a projection of the cut* $(S, V \setminus S)$ *on* U. *Further, we will say that a cut* $(C, U \setminus C)$, *where* $C \subseteq U$, *is a projected cut of conductance* Φ *if it is a projection of a cut* $(S, V \setminus S)$ *with conductance* Φ.

5.1.1 Contributions of this study

Our approach builds on the streaming algorithms presented in [48] for performing a large number of random walks efficiently on a graph stream. These random walks are

used to estimate the probability distribution of the random walk that is in turn be used to find a sparse cut by adapting the method of Lovasz and Simonovits [113, 136].

One of the main contributions of this work is an algorithm to estimate the probability distributions on an arbitrarily chosen subset of k nodes in a d-regular graph. To obtain the probability of reaching destination t from source s after a walk of length l, the algorithm runs multiple walks (starting with length $l/2$) from source-destination pairs, and recursively estimates probability distributions of mid-points, by looking at the "collisions" of these walks. A similar idea has been used in property testing for expander graphs in [75]. However, in their case, they just need to run walks of length $l/2$ and investigate the collisions. Since we need a good estimate of the probability distribution at t, the algorithm needs to run walks recursively of shorter lengths. All our techniques depend on the reversibility of the random walk, and hence only work for d-regular, unweighted graphs. We now describe our results beginning with a definition of some notation.

Definition 5.1.4. *Let $P_l[st]$ denote the probability of landing at node t after a random walk of length l starting from s. Further, let $p_l(i) = P_l[si]$. We drop the subscript l when it is clear from context.*

The following theorem, proved in Section 5.3, shows how to compute the approximate distribution on a arbitrarily chosen subset K of k nodes.

Theorem 5.1.5. *Given an arbitrarily chosen subset K of k nodes, one can compute an estimate $\tilde{p}(i)$ for $p(i)$ (the probability distribution after a walk of length l) for all $i \in K$ in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{nkl}{\alpha}})$ space for any choice of $\alpha \leq 1$, such that the error in the estimate $|\tilde{p}(i) - p(i)|$ is at most $\tilde{O}(l\sqrt{\frac{p(i)\epsilon}{n}} + \frac{l\epsilon}{n} + (l\sqrt{\epsilon})p(i))$.*

Our main results for computing projected cuts (described in Section 5.4) with sparsity at most $\tilde{O}(\sqrt{\Phi})$ are stated below.

Theorem 5.1.6. *For any d-regular graph G that has a cut of balance b and conductance at most Φ, given a set K of randomly chosen k nodes, we show that there is an algorithm that achieves the following on a graph stream (for any choice of $\alpha \leq 1$).*
(a) Partitions K into two sets such that the partitioning is a projected cut of conductance at most $\tilde{O}(\sqrt{\Phi})$, in $\tilde{O}(\frac{1}{\sqrt{\alpha\Phi}})$ passes and $\tilde{O}(n\alpha + \frac{n^{3/4}k^{1/4}}{b\sqrt{\alpha}\Phi^{19/4}})$ space.
(b) Outputs k candidate partitions such that at least one of them is a projected cut of conductance at most $\tilde{O}(\sqrt{\Phi})$, in $\tilde{O}(\frac{1}{\sqrt{\alpha\Phi}})$ passes and $\tilde{O}(n\alpha + \frac{\sqrt{nk}}{b\sqrt{\alpha}\Phi^{\frac{9}{2}}})$ space.

Corollary 5.1.7. *Given a set of randomly chosen $k \leq n\alpha^6 b^4 \Phi^{O(1)}$ nodes, there is an algorithm that partitions them, in $\tilde{O}(\frac{1}{\sqrt{\Phi\alpha}})$ passes and $\tilde{O}(n\alpha)$ space, into a projected cut of conductance at most $\tilde{O}(\sqrt{\Phi})$ w.h.p.*

Observe that the space required is sublinear in the number of nodes if k satisfies the bound in the above corollary. Our algorithms can also be extended to partition all the n nodes in the graph; that is, find the entire (approximate, sparse) cut. The following theorem shows how find an approximate sparse cut in (possibly) sublinear space. The proof is detailed in Section 5.5, for ease of presentation.

Theorem 5.1.8. *For any d-regular graph G that has a cut of conductance at most Φ and balance b, there is an algorithm that performs $\tilde{O}(\sqrt{\frac{1}{\Phi\alpha}})$ passes over the graph stream and using space $\tilde{O}(\min\{n\alpha + \frac{1}{b}\left(\frac{n\alpha}{d\Phi^3} + \frac{n}{d\sqrt{\alpha}\Phi^{5/2}}\right), (n\alpha + \frac{1}{b}\frac{n}{d\alpha\Phi^2})\sqrt{\frac{1}{\Phi\alpha}} + \frac{1}{\Phi}\})$, for any choice of $\alpha \leq 1$. and outputs, with high probability, a cut of conductance at most $\tilde{O}(\sqrt{\Phi})$.*

5.1.2 Related Work

A well-known approach for graph partitioning is to compute the second eigenvector that can be used to compute a sparse cut by ordering the nodes in increasing order of coordinate value in the eigenvector. The second eigenvector technique has been analyzed in a series of results [8, 28, 135] relating the gap between the first and second eigenvalue.

The best known approximation algorithm to compute the sparsest cut in a graph is due to Arora, Rao, and Vazirani [13]. They provide $O(\sqrt{\log n})$-approximation algorithm using semi-definite programming techniques. While their algorithm guarantees good approximation ratios, it is slower than algorithms based on spectral methods and random walks.

Lovasz and Simonovits [113, 112] proposed another approach to finding cuts of small conductance. They showed how random walks can be used to find sparse cuts. Specifically, they show that if you start a random walk from a certain node and order the nodes by the probability of reaching them, then this ordering contains a sparse cut. They prove that if the sparsest cut has conductance ϕ, then their method can be used to find a cut with conductance at most $O(\sqrt{\phi})$.

Spielman and Teng [136] build upon the work of Lovasz and Simonovits and show how it can be implemented more efficiently by sparsifying the graph. They show that for a dense graph, it is possible to look at a near linear number of edges and only compute the sparse cuts on the sampled set of vertices. Given a graph $G = (V, E)$ with a cut $(S, V \setminus S)$ with sparsity ϕ and balance $b(S) = |e(S)|/2|E| \geq 1/2$ where $e(.)$ denotes the set of edges incident on nodes in S, their algorithm finds a cut $(D, V \setminus D)$ with sparsity $O(\phi^{1/3} \log^{O(1)} n)$ and balance of the cut $(D, V \setminus D)$, $b(D) \geq b(S)/2$.

Andersen, Chung, and Lang [10] proposed a local partitioning algorithm to find cuts near a specified vertex and global cuts. The running time of their algorithm was proportional to the size of small side of the cut. Their results improve upon those in [136];

In a more recent work [48], the authors proposed algorithms to perform several random walks efficiently on graphs presented as edge streams using a small number of passes. A recent study [4] shows how to find $1+\epsilon$-approximate sparse cuts in $\tilde{O}(n)$ space by making use of graph sparsifiers. In contrast, our algorithm requires sublinear space for a certain range of parameters, but provides much a weaker approximation

to the sparsest cut.

5.2 Cuts from approximate probability distributions of random walks

In this section we will show how one can compute candidate sparse cuts from approximate probability distributions of random walks. We start from a random source s from the smaller side of the best cut with conductance Φ and perform a random walk of length about $1/\Phi$. We extend the algorithm of Lovasz and Simonovits [113] to find sparse cuts using approximate distributions. This is similar to the work by Spielman and Teng [136] that also works with estimates of $p(i)$. But the magnitude of error allowed in our work is larger than in theirs. We adapt a set of lemmas from their work to prove Theorem 5.2.2 below. The proof is detailed in Section 5.5 at the end of this chapter.

Definition 5.2.1. *For a probability distribution $p(i)$ on nodes, let $\rho_p(i) = p(i)/d(i)$. Let π_p denote the ordering of nodes in decreasing order of ρ_p; that is, $\rho_p(\pi_p(i)) \geq \rho_p(\pi_p(i+1))$.*

Recall that $p(i)$ denotes the probability of ending at node i after a random walk of length l. The following theorem shows how one can find candidate sparse cuts using approximate values $\tilde{p}(i)$ of $p(i)$. It looks at the n candidate cuts obtained by ordering the nodes in the order $\pi_{\tilde{p}}$.

Theorem 5.2.2. *Let $(U, V \setminus U)$ (with $|U| \leq |V|/2$) be a cut of conductance at most Φ. Let $\tilde{p}(i)$ denote an estimate for the probability $p(i)$ of a random walk of length l from a source s from U. Assume that $|\tilde{p}(i) - p(i)| \leq \epsilon(p(i) + 1/n)$, where $\epsilon \leq o(\Phi)$. Consider the n candidate cuts obtained by ordering the vertices in decreasing order of $\rho_{\tilde{p}}(i)$; each candidate cut $(S, V \setminus S)$ is obtained by setting S equal to a prefix $S_j = \pi_{\tilde{p}}\{1, 2, \ldots, j\}$. If the source node s is chosen randomly from U and the length l is chosen randomly*

in the range $\{1, \ldots, O(1/\Phi)\}$*, then with constant probability, one of these* n *candidate cuts has conductance* $\Phi(S_j) \leq \tilde{O}(\sqrt{\Phi})$

Note that the source node s needs to be sampled from U, the smaller side of the cut. To obtain such a source, we have to sample several sources from V, since U is not known, and execute the algorithm in parallel (so as not to increases the number of passes required). Given a cut of balance b, this increases the number of walks required by a factor of $\tilde{O}(\frac{1}{b})$, and therefore the space required accordingly; in all our space bounds, the first term of $n\alpha$, however, does not depend on the number of walks performed.

In section 5.4 we will show how Theorem 5.2.2 in conjunction with Theorem 5.1.5 is used to prove Theorem 5.1.6. The essential idea is to look at the k candidate cuts obtained by arranging the nodes in decreasing order of $\pi_{\tilde{p}}$ and then estimate the conductance across these candidate cuts to pick the best one.

In proving these theorems, we use the techniques presented in [48] for performing a large number of random walks efficiently on a graph stream. They show how to perform $O(n/l)$ independent random walks using $\tilde{O}(n\alpha)$ space and $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes over the graph stream. Their main result is stated below.

Theorem 5.2.3 ([48])**.** *One can perform* k *independent random walks from a given source distribution, on a graph stream, in* $\tilde{O}(\sqrt{l/\alpha})$ *passes and* $\tilde{O}(\min\{n\alpha + kl\alpha + k\sqrt{l/\alpha},$
$n\alpha\sqrt{l/\alpha} + k\sqrt{l/\alpha} + l\})$ *space, for any choice of* $\alpha \leq 1$*. For* $k = \frac{n}{l}$ *walk, this requires* $\tilde{O}(\sqrt{l/\alpha})$ *passes and* $\tilde{O}(n\alpha)$ *space for* $1/l \leq \alpha \leq 1$*.*

Next we will show how performing random walks can be used to compute the probability distribution approximately so that we may apply theorem 5.2.2. To get good approximations, we need to perform random walks recursively as shown in the next section.

5.3 Estimating probability distribution p_i on a small set of nodes

In this section we show how to estimate the probability distribution on a small set of k nodes resulting in Theorem 5.1.5. The distribution is required for the endpoint of a random walk of length l from a specific source node s (or more generally a source distribution). The naïve approach would be to perform several random walks of length l from s and look at the end points of these walks to see how many times each of the k nodes occurs. This can be inefficient as k may be much smaller than n and most of the random walks may end up at nodes other than the k nodes we are interested. So we seek a more efficient approach tailored towards estimating the distribution of a specific small set of nodes.

We will begin by stating the following technical lemma. The lemma is later used to approximate distributions. It bounds the error in estimating a_{ij} for a matrix A, where i and j are drawn from two different probability distributions. The guarantee is stated as a trade-off with the number of samples drawn for i and for j.

Lemma 5.3.1. *Let $A = \{a_{ij}\}_{m\times n}$ denote a matrix of non-negative entries a_{ij}. Let $\mu_{XY} = E_{i\in X, j\in Y}[a_{ij}]$ denote the expected value of a_{ij} where i and j are drawn independently from probability distributions $X = \{x_i, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ on the rows and columns respectively. Assuming $||A^t x||_\infty \leq \tilde{O}(\frac{1}{\epsilon n_x})$ and $||Ay||_\infty \leq \tilde{O}(\frac{1}{\epsilon n_y})$, one can obtain an estimate $\mu_{\tilde{X}\tilde{Y}}$ for μ_{XY} by drawing $\tilde{O}(n_x)$ samples from X and $\tilde{O}(n_y)$ samples from Y. Here $\tilde{X}$ and $\tilde{Y}$ are the distributions induced by the $\tilde{O}(n_x)$ and $\tilde{O}(n_y)$ samples respectively. The error $|\mu_{\tilde{X}\tilde{Y}} - \mu_{XY}|$ is at most $\tilde{O}(\sqrt{\frac{\mu_{XY}}{\epsilon n_x n_y}} + \frac{1}{\epsilon n_x n_y})$ w.h.p.*

Proof. Let $\mu_D = E_{i\in D}[c_i]$. For a distribution D and a vector c with non-negative entries between $[0, 1]$, $\tilde{O}(n)$ samples are drawn from D to estimate μ_D w.h.p such that $|\mu_{\tilde{D}} - \mu_D| \leq \sqrt{\frac{\mu}{n}} + \frac{1}{n}$ by Chernoff bounds. More generally, $|\mu_{\tilde{D}} - \mu_D| \leq \sqrt{\frac{\mu||c||_\infty}{n}} + \frac{||c||_\infty}{n}$.

We need to bound $|\mu_{XY} - \mu_{\tilde{X}\tilde{Y}}| \leq |\mu_{XY} - \mu_{X\tilde{Y}}| + |\mu_{\tilde{X}Y} - \mu_{\tilde{X}\tilde{Y}}|$. Set $c_i = E_{j \in Y}[a_{ij}]$, i.e., $c = Ay$. This gives $|\mu_{XY} - \mu_{\tilde{X}Y}| \leq \sqrt{\frac{\mu_{XY}||Ay||_\infty}{n_x}} + \frac{||Ay||_\infty}{n_x} \leq \sqrt{\frac{\mu_{XY}}{\epsilon n_x n_y}} + \frac{1}{\epsilon n_x n_y}$.

Further, setting $c_j = E_{i \in \tilde{X}}[a_{ij}]$ or $c = A^t\tilde{x}$ gives $|\mu_{\tilde{X}Y} - \mu_{\tilde{X}\tilde{Y}}| \leq \sqrt{\frac{\mu_{\tilde{X}Y}||A^t\tilde{x}||_\infty}{n_y}} + \frac{||A^t\tilde{x}||_\infty}{n_y}$ w.h.p. But, note that $||A^t\tilde{x}||_\infty \leq ||A^tx||_\infty + \tilde{O}(\sqrt{\frac{A^t||x||_\infty}{n_x}} + \frac{A^t||x||_\infty}{n_x}) \leq \tilde{O}(\frac{1}{\epsilon n_x})$ as $||A^tx||_\infty \leq \tilde{O}(\frac{1}{\epsilon n_x})$. And since $\mu_{\tilde{X}Y} \leq \mu_{XY} + \tilde{O}(\sqrt{\frac{\mu_{XY}}{\epsilon n_x n_y}} + \frac{1}{\epsilon n_x n_y}) \leq \tilde{O}(\mu_{XY} + \frac{1}{\epsilon n_x n_y})$, the difference is at most $\tilde{O}(\sqrt{\frac{(\mu_{XY} + \frac{1}{\epsilon n_x n_y})1/\epsilon n_x}{n_y}} + \frac{1}{\epsilon n_x n_y}) \leq$
$\tilde{O}(\sqrt{\frac{\mu_{XY}}{\epsilon n_x n_y}} + \frac{1}{\epsilon n_x n_y})$. Combining the two, the lemma follows. ∎

□

We first describe the main idea in estimating the probabilities after a random walk for a subset of nodes as Algorithm RecursiveRandomWalk. The algorithm uses a parameter m that controls the accuracy of error in estimation. Given a set of nodes K, with $|K| = k$, and a source node s, we wish to estimate $P_l[st]$ for all nodes in $t \in K$ up to an additive accuracy of about $O(\sqrt{P_l[st]/m})$. Rather than performing m walks of length l from s, $\tilde{\Theta}(\sqrt{mk})$ walks are performed from s and $\tilde{\Theta}(\sqrt{\frac{m}{k}})$ walks from each node in K, all of length $l/2$. Note that the product of the number of walks performed is m. We then use collisions in the end points of these walks of length $l/2$ to estimate the probabilities (since it is a reversible random walk). The key insight is that $P_l[st] = \sum_i P_{l/2}[su_i].P_{l/2}[tu_i] = \sum_i x_i y_i$ where $x_i = P_{l/2}[su_i]$ and $y_i = P_{l/2}[tu_i]$. That is, one can break all paths from s to t at half way, and sum over all $l/2$ length paths from s to u_i and u_i to t. Any node u_i may either have a *small* or a *large* probability of being reached from s after a random walk of length $l/2$; the same observation holds for for t as well. Roughly, a node u has a small probability for source s if $P_{l/2}[su]$ is $o(1/\sqrt{mk})$, and large probability otherwise. In the formal description of the algorithm we denote these sets of nodes by S_a and S_b respectively. Notice that $o(1/\sqrt{mk})$ is less than the reciprocal of the number of walks run from s. We denote the number of walks of length $l/2$ performed from s by n_s. Similarly,

a node u has small probability estimate from t if $P_{l/2}[tu]$ is $o(\sqrt{m/k})$, and a large probability otherwise. In the algorithm, these sets of nodes are denoted by T_a and t_b respectively. The total number of walks of length $l/2$ performed from each node t is denoted by n_t. Now, four cases arise for every u_i.

- $x_i y_i$ is $\Omega(1/m)$ and x_i is $\Omega(\frac{1}{\sqrt{mk}})$ and y_i is $\Omega(\sqrt{\frac{k}{m}})$.
- $x_i y_i$ is $o(1/m)$ and x_i is $o(\frac{1}{\sqrt{mk}})$ and y_i is $o(\sqrt{\frac{k}{m}})$.
- $x_i y_i$ is $\Omega(1/m)$ but x_i is $o(\frac{1}{\sqrt{mk}})$ and y_i is $\Omega(\sqrt{\frac{k}{m}})$.
- $x_i y_i$ is $\Omega(1/m)$ but x_i is $\Omega(\frac{1}{\sqrt{mk}})$ and y_i is $o(\sqrt{\frac{k}{m}})$.

The first case is when u_i has large $P_{l/2}[su_i]$ and $P_{l/2}[tu_i]$, and therefore, it will be seen in walks from both ends and gives us a good estimate of u_i's contribution to $\sum_i P_{l/2}[su_i].P_{l/2}[tu_i]$. The second case is when u_i has small probability for both s and t. In this case, w.h.p., u_i will not be seen in either set of walks. Therefore, u_i's contribution to $\sum_i P_{l/2}[su_i].P_{l/2}[tu_i]$ cannot be estimated. However, since $P_{l/2}[su_i].P_{l/2}[tu_i]$ itself is $o(1/m)$, u_i's contribution to the estimate of $P_l[st]$ is *negligible*.

The difficulty arises in estimating the product for u_i in the third and fourth cases. In both these scenarios, just the walks described above aren't sufficient to estimate the product and yet the contribution may be significant. This is because u_i has a large probability from one end, but a small probability from the other end. The small probability cannot be estimated with just these walks, but the product could be significant, in particular the product could be $\Omega(1/m)$. Hence one needs to resort to a recursive estimation algorithm where the small estimate is captured by performing further walks.

For any node u_i with a large value of $P_{l/2}[su_i]$ and a small value of $P_{l/2}[tu_i]$, we adopt a recursive approach between u_i and t by performing random walks of length $l/4$ from all such u_i and from t. Similarly, for all nodes u_i that have a large value of $P_{l/2}[tu_i]$ and a small value of $P_{l/2}[su_i]$, we perform random walks of length $l/4$ from s and all these u_i to get better estimates of $P_{l/2}[su_i]$ and consequently the product $P_l[st]$. These probabilities may themselves be estimated by random walks of length

Algorithm 16 RecursiveRandomWalk(s, K, l)

1: **Input:** Starting node/distribution s, length l, and chosen set of k nodes K. Set K need not necessarily be chosen at random.
2: **Output:** $p(t) = \tilde{P}_l[st]$ for each $t \in K$, an estimate of $P_l[st]$ with explicit bound on additive error.
3: Perform $n_s = \tilde{\Theta}(\sqrt{mk})$ walks from s and $n_t = \tilde{\Theta}(\sqrt{m/k})$ walks from each $t \in K$, all of length $l/2$.
4: Denote by S_a the set of nodes seen at most $\tilde{O}(\frac{1}{\epsilon})$ times (small number of times) as endpoints of the n_s walks from s and denote the remaining nodes (seen large number of times) by S_b. Similarly, for each t, partition the nodes into t_a(seen small number of times from t) and t_b(seen large number of times from t). Denote by $\tilde{x}$ and $\tilde{y}$ the distributions of nodes in the end points of the walks from s and t respectively. Thus, $\tilde{x}_i$ is the fraction of walks from s that end up at node i.
5: Let $w(S_b) = \sum_{i \in S_b} \tilde{x}_i$ and $w(t_b) = \sum_{i \in t_b} \tilde{y}_i$. Denote by D_{S_b} the distribution of end points of walks over the nodes in S_b, i.e., the probability of i in D_{S_b} is $\tilde{x}_i / w(S_b)$ if $i \in S_b$ and 0 otherwise. Similarly, denote by D_{t_b} the distribution of end points of walks over the nodes in t_b.
6: For each $t \in K$, set $P_l[st] = \sum_{i \in S_a \cap t_a} \tilde{x}_i \tilde{y}_i + w(S_b) P_{l/2}[D_{S_b} t] + w(t_b) P_{l/2}[s D_{t_b}] - \sum_{i \in S_b \cap t_b} \tilde{x}_i \tilde{y}_i$.
7: In the above expression, $P_{l/2}[D_{S_b} t]$ and $P_{l/2}[s D_{t_b}]$ are estimated recursively for all the $t \in K$. Note that if length is 1, $P_1[st]$ can be computed exactly in one pass.

$l/8$, in another depth of the recursion, and so on. Eventually, combining all of these carefully gives us the probability distribution of t from s after a random walk of length l (notice that we use the reversibility of the random walk). The exact details are stated in Algorithm 16.

We now state and prove the guarantee of RecursiveRandomWalk in estimating probabilities by making use of Lemma 5.3.1.

Lemma 5.3.2. *Algorithm* RecursiveRandomWalk *gives an estimate of* $\mu = P_l[st]$ *within an additive error of* $\tilde{O}(l\sqrt{\epsilon}\mu + l\sqrt{\frac{\mu}{\epsilon m}} + \frac{l}{\epsilon m})$.

Proof. $P_l[st] = \sum_{i=1}^{n} x_i y_i$. If all x_i are smaller than $\tilde{O}(\frac{1}{\epsilon\sqrt{km}})$ and y_i is smaller than $\tilde{O}(\frac{1}{\epsilon\sqrt{m/k}})$, then by Lemma 5.3.1, the algorithm estimates $\mu = P_l[st]$ within error of $\tilde{O}(\sqrt{\frac{\mu}{\epsilon m}} + \frac{1}{\epsilon m})$. More generally, $P_l[st]$ can be computed as a sum over four sets as $P_l[st] = \sum_{i \in S_a \cap t_a} x_i y_i + \sum_{i \in t_b} x_i y_i + \sum_{i \in S_b} x_i y_i - \sum_{i \in S_b \cap t_b} x_i y_i$; here $i \in S_a$ if x_i is $\tilde{O}(\frac{1}{\epsilon\sqrt{km}})$ and $i \in S_b$ otherwise. $-\sum_{i \in S_b \cap t_b} x_i y_i$ is required as it is counted once in each

of $\sum_{i\in t_b} x_i y_i$ and $\sum_{i\in S_b} x_i y_i$. Similarly $j \in t_a$ if y_j is $\tilde{O}(\frac{1}{\epsilon\sqrt{m/k}})$, and $j \in t_b$ otherwise. We will argue that step 6 of the algorithm RecursiveRandomWalk is summing the estimates of each term.

Let $\mu_{aa}, \mu_{*b}, \mu_{b*}, \mu_{bb}$ respectively denote $\sum_{i\in S_a\cap t_a} x_i y_i$, $\sum_{i\in t_b} x_i y_i$, $\sum_{i\in S_b} x_i y_i$ and $\sum_{i\in S_b\cap t_b} x_i y_i$. Note that it is not known which of x_i's or y_i's are small or large. The number of walks performed, however, are sufficient to obtain the right classification to small or large, by standard Chernoff bounds.

Let $\tilde{x}$ and $\tilde{y}$ denote the distributions induced by the end points of the n_s walks and n_t walks respectively. Note that by Lemma 5.3.1, $\sum_{i\in S_a\cap t_a} x_i y_i$ can be estimated as $\sum_{i\in S_a\cap t_a} \tilde{x}_i\tilde{y}_i$ with error at most $\tilde{O}(\sqrt{\frac{\mu_{aa}}{\epsilon m}} + \frac{1}{\epsilon m})$. Also note that if $i \in S_b$, then $\tilde{x}_i$ is within $(1 \pm \sqrt{\epsilon})\tilde{x}_i$ w.h.p. from Chernoff bounds. Thus, μ_{bb} can be estimated with error at most $\sqrt{\epsilon}\mu_{bb}$. Again, μ_{*b} can be estimated as $\mu_{*\tilde{b}} = \sum_{i\in t_b} x_i\tilde{y}_i$ where the error $|\mu_{*b} - \mu_{*\tilde{b}}| \le \sqrt{\epsilon}\mu_{*b}$. Similarly, μ_{b*} can be estimated as $\mu_{\tilde{b}*} = \sum_{i\in S_b} \tilde{x}_i y_i$ with additive error at most $\sqrt{\epsilon}\mu_{b*}$.

Observe that the $\mu_{*\tilde{b}} = \sum_{i\in t_b} x_i\tilde{y}_i = w(S_b)P_{l/2}[D_{S_b}t]$ and $\mu_{b\tilde{*}} = \sum_{i\in S_b} \tilde{x}_i y_i = w(t_b)P_{l/2}[sD_{t_b}]$ are estimated recursively by computing $P_{l/2}[D_{S_b}t]$ and $P_{l/2}[sD_{t_b}]$. Let $\delta_l(P_l[st])$ denote the error in estimating the probability $P_l[st]$. Then $\delta_l(\mu) = \sqrt{\frac{\mu_{aa}}{\epsilon m}} + \frac{1}{\epsilon m} + \sqrt{\epsilon}\mu_{bb} + \sqrt{\epsilon}\mu_{b*} + \sqrt{\epsilon}\mu_{*b} + w(S_b)\delta_{l/2}(P_{l/2}[D_{S_b}t]) + w(t_b)\delta_{l/2}(P_{l/2}[sD_{t_b}]) \le \sqrt{\frac{\mu}{\epsilon m}} + \frac{1}{\epsilon m} + 3\sqrt{\epsilon}\mu + w(S_b)\delta_{l/2}(\frac{\mu}{w(S_b)}) + w(t_b)\delta_{l/2}(\frac{\mu}{w(t_b)})$.

The branching factor of the recursion is 2 and has depth $\log l$ with l leaves. Also note that at the leaf, $\delta_1(P_1[st]) = 0$. It follows from standard methods for solving recursion that $\delta_l(\mu) = \tilde{O}(l\sqrt{\epsilon}\mu + l\sqrt{\frac{\mu}{\epsilon m}} + \frac{l}{\epsilon m})$. ∎

□

We now use the approach in [48] to bound the number of passes and space required in performing RecursiveRandomWalk. The analysis is simple and only requires a careful calculation of the number of walks performed for each length $l/2, l/4, l/8, \ldots$.

Lemma 5.3.3. *Algorithm* RecursiveRandomWalk *can be implemented on a graph stream in* $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ *passes and* $\tilde{O}(n\alpha + \sqrt{\frac{mkl}{\alpha}})$ *space for any choice of* $\alpha \leq 1$.

Proof. Notice that in the first phase of RecursiveRandomWalk, $O(\sqrt{mk})$ walks are performed from s of length $l/2$ and $O(\sqrt{m/k})$ walks from each t of length $l/2$. Whenever recursively calculating the distribution, $O(\sqrt{mk})$ walks are required for any source distribution to destination distribution pair. Since the length of the walks halve with every recursive depth, the number of levels is $O(\log l)$ before the length of the walks required becomes a constant. However, the pairs for which the distributions need to be estimated keeps doubling. So after, say, t phases, we perform $O(2^t\sqrt{mk})$ walks of length $l/2^t$.

From [48], the space required for k walks is $\tilde{O}(n\alpha + kl\alpha + k\sqrt{l/\alpha})$. Although [48] assumes that the source distribution was the same for all the k walks, it is easy to extend their result to perform a specific number of walks from different source distributions (to a total of k walks), with only a logarithmic factor increase in the space (by Chernoff bounds).

Summing this over t phases, the first term remains $\tilde{O}(n\alpha)$. The second term of $\tilde{O}(kl\alpha)$ is always $\tilde{O}(\sqrt{mk}l\alpha)$, as only a $\log l$ factor increases. The third term of $k\sqrt{l/\alpha}$ is dominated by the last phase (since this term depends linearly in k but only as square-root in the length of the walks), where $l\sqrt{mk}$ walks of length $O(1)$ are performed. The dominating term therefore is $\tilde{O}(\sqrt{mk}l\alpha)$, completing the proof. ∎

□

Remark 5.3.4. *If algorithm* RecursiveRandomWalk *is to be performed from* r *different sources, this would increase the space requirement to* $\tilde{O}(n\alpha + r\sqrt{\frac{mkl}{\alpha}})$. *This follows from the fact that the first term of* $\tilde{O}(n\alpha)$ *in the space requirement does not depend on the number of walks performed.*

Notice that combining Lemmas 5.3.2, 5.3.3, and choosing $m = \frac{n}{\epsilon^2}$ immediately

gives Theorem 5.1.5. Observe that by setting $\epsilon = o(\Phi^2/l^2)$ we satisfy the conditions of Theorem 5.2.2 and can thus compute candidate cuts.

5.4 Finding sparse cut projections on a small set of nodes

In this section, we prove Theorem 5.1.6. Approximate values of $p(i)$ are known from Theorem 5.1.5. By setting $\epsilon = \tilde{O}(\frac{\Phi^2}{l^2})$ in theorem 5.2.2, we find probability estimates $\tilde{p}(i)$ that satisfy the condition required in theorem 5.1.5. Notice that part (b) of Theorem 5.1.6 follows directly by using walks of length $l = O(\frac{1}{\Phi})$. If we order all the n vertices by the probability estimates at least one cut has conductance at most $\tilde{O}(\sqrt{\Phi})$. Naturally this ordering induces an ordering on the k vertices that results in k candidate cuts. Note that in our algorithm we need to sample $\tilde{\Omega}(1/b)$ sources so that at least one falls on the smaller side of the optimal cut with high probability. The factor $1/b$ is not applied to the $n\alpha$ term as we can perform all the random walks concurrently as stated in remark 5.3.4.

We now prove Theorem 5.1.6 part (a). We need to estimate the projected cut conductance for each of the k candidate cuts from the ordering of approximate probabilities. This is done by boosting the number of random nodes from k to k'. It turns out that one can estimate the projected cut conductance value of a specific cut on the k nodes by looking at the conductance on the induced subgraph and cuts on the k' nodes (for an appropriate choice of k'), as stated in Lemma 5.4.1. The formal description is in algorithm CutProjectionCandidates and algorithm CutProjection. Let $\Phi_{K'}(U, K' \setminus U)$ denote the conductance of the cut $(U, K' \setminus U)$ on the induced subgraph on nodes in K'.

Lemma 5.4.1. *For a set K' of randomly chosen nodes with $|K'| = k' \geq \sqrt{\frac{kn}{\phi}}$ and $|U| \geq \frac{k'}{k}$ and $|K' \setminus U| \geq \frac{k'}{k}$, and let $(U, K' \setminus U)$ be a projected cut of conductance of be Ψ. Then $\Phi_{K'}(U, K' \setminus U)$ gives a constant factor approximation to Ψ with high probability.*

Algorithm 17 CutProjectionCandidates(G, K, s)

1: **Input:** Graph G, set K of k randomly sampled nodes, and a source node s.
2: **Output:** k partitions on these k nodes.
3: Estimate probabilities on the k nodes using RecursiveRandomWalk for source s and walk of length l, where l is chosen at random between 1 and $O(1/\phi^2)$.
4: Order the k nodes in decreasing order of the probability estimates. Return the k candidate cuts implied by taking prefixes of this ordering.

Algorithm 18 CutProjection

1: **Input:** Graph G with cut of conductance at most Φ and source s from the smaller side of this cut; set K of k randomly sampled nodes.
2: **Output:** Partition of these k nodes such that this is a projected cut of conductance is at most $\phi = \tilde{O}(\sqrt{\Phi})$ with constant probability.
3: Sample additional nodes randomly and add it to the set K so that the resulting set K' is of size $k' = \sqrt{\frac{kn}{\phi}}$.
4: Call CutProjectionCandidates with G', K', s.
5: Consider all of the k' cuts returned by CutProjectionCandidates that have at least $\frac{k'}{k}$ nodes on either side of the cut. Notice that each of these cuts has at least one of the k nodes in K on either side, with constant probability.
6: For each of these cuts, compute the conductance on the induced subgraph over these k' nodes.
7: Output the cut induced on the k nodes by the cut on the k' nodes that has the minimum conductance in the induced subgraph.

sketch. The essential idea is that considering the conductance of the induced cut on $\sqrt{\frac{kn}{\phi}}$ nodes, is identical to estimating $\sum_{(i,j)\in E} na_{ij}x_iy_j/(|X||Y|)$ by sampling each x_i with probability $k'|X|/n$ and each y_i with probability $k'|Y|/n$ and a_{ij} is $\frac{1}{d}$ for an edge (i,j). The proof is then completed using Lemma 5.3.1. A more detailed exposition is presented in Section 5.5. ∎

□

This lemma automatically gives a method for estimating the projected cut conductance for a partition of a subset of k nodes. We are now ready to prove the main theorem 5.1.6 part (b).

Proof of Theorem 5.1.6(b). By setting $\epsilon = \tilde{O}(\frac{\Phi^2}{l^2})$ in theorem 5.2.2, we find probability estimates $\tilde{p}(i)$ that satisfy the condition required in theorem 5.1.5. So if we

order all the n vertices by the probability estimates at least one cut has conductance at most $\tilde{O}(\sqrt{\Phi})$. Naturally this ordering induces an ordering on the k' vertices that contain the set K. We are simply estimating the conductance of all the cuts in this ordering that has at least one vertex from K in the smaller side. If none of these cuts give the required conductance of $\tilde{O}(\sqrt{\Phi})$, then all k nodes are put on the same side of the cut and output. Note that in our algorithm we need to sample $\tilde{\Omega}(1/b)$ sources so that at least one falls on the smaller side of the optimal cut with high probability. This gives $\sqrt{\frac{l}{\Phi\alpha}}$ passes and $\tilde{O}(n\alpha + \frac{1}{b}\frac{1}{\epsilon}\sqrt{\frac{nk'}{\phi\alpha}}) = \tilde{O}(n\alpha + \frac{n^{3/4}k^{1/4}}{b\sqrt{\alpha}\Phi^{19/4}})$ space. The factor $1/b$ is not applied to the $n\alpha$ term as the we can perform all the random walks concurrently as stated in remark 5.3.4. ∎

□

5.5 Full Proofs

5.5.1 Cuts from Approximate Distributions

In this section, we will prove Theorem 5.2.2. We use Φ to denote the conductance of the graph G. We use ϕ to denote the conductance of the approximate cut obtained by our algorithm.

We first state the algorithm used to find these cuts, which is a slight modification of the algorithm in [136]. The basic idea is to first sample a source node, say s. If we are after a cut with conductance Φ, say $(U, V \setminus U)$ and say $|U| \leq |V \setminus U|$, then the hope is that s is from the smaller side U. Like the section on find projected cuts, here as well, we can sample a number of sources, starting from $\log n$, and doubling till $1/b$ if required. So for this section, assume that we have sampled the source from the smaller side (which is equivalent to assuming balance of a sparse cut). We then choose l at random from the range of 1 to $1/\Phi$; this is again repeated for $\log n$ different trials for every s. Random walks of length l are performed from s, to estimate the probability distributions of all nodes from source s after a walk

of length l. This quantity is denoted by $P_l[sv]$ for node v. The nodes are then ordered in decreasing order of probabilities, and all n cuts consistent with this ordering are tried to see which has minimum conductance. For one value of s, the algorithm guarantees a quadratic approximation to Φ, with constant probability, if the probability estimates have sufficiently high accuracy. The algorithm is formally described in CutsFromDistributions.

Remark 5.5.1. CutsFromDistributions *samples a source from the smaller side of the cut in* $O(\log n)$ *source samples, since we assume that the input graph* G *has a cut of conductance* Φ *with constant balance.*

For this section, we assume that our probability estimation techniques give the following bound on the approximate probability of every vertex i.

$$|\tilde{p}(i) - p(i)| \quad \leq \quad \epsilon(p(i) + 1/n)$$

This is equivalent to assuming $|\tilde{p}(i) - p(i)| \leq \epsilon(p(i) + \sqrt{p(i)/n} + 1/n)$ since the central term is the geometric mean. Here, $\tilde{p}(i)$, as before, denotes the estimate of $p(i)$ by our algorithms. We follow the approach of Spielman and Teng [136] to finding cuts, and modify some of their lemmas to suit our probability estimates. First we will define some of the notation used in their paper. We use T to denote the transition probability matrix of the graph we are considering. So given a probability distribution p, Tp gives the probability distribution after a random walk of length one from the source distribution p. Further, we need one last definition of a function H that depends on a probability distribution p and the ordering π_p.

Definition 5.5.2. *Let* $k_j^p = \sum_{i=1}^{j} d(\pi_p(i))$. *Then,* $H_p(k_j^p) = \sum_{i=1}^{j} (p(\pi_p(i)) - d(\pi_p(i))/2m)$.

H_p is then made piece-wise continuous, by joining the above points with line segments. In certain contexts, we use H_l to denote H_{p_l}.

Lemma 5.5.3 ([136])**.** $H_{Tp}(x) \leq H_p(x)$ *for any probability distribution* p *and all* x.

Algorithm 19 CUTSFROMDISTRIBUTIONS

1: **Input:** Graph G with a cut of conductance Φ of constant balance. Algorithm to estimate probabilities up to additive error $\epsilon_1 p_l(v) + \sqrt{\epsilon_2 p_l(v)/n} + \epsilon_3/n$.
2: **Output:** Partition of n nodes such that the cut has conductance at most $\tilde{O}(\sqrt{\Phi})$.
3: Chose a source s at random.
4: Estimate the conductance of the graph using the technique in [48].
5: Choose l between 1 and $\frac{log 1/\epsilon}{\Phi}$ at random.
6: Estimate $P_l[sv]$ for all v using the input algorithm, up to an additive error of at most $\epsilon_1 p_l(v) + \sqrt{\epsilon_2 p_l(v)/n} + \epsilon_3/n$
7: Order the nodes in decreasing order of $\rho_{\tilde{p}}(i)$ and choose the cut with minimum conductance among the n cuts consistent with this ordering.
8: This cut has conductance at most $\tilde{O}(\sqrt{\Phi})$ with constant probability. Amplify the probability by sampling l $O(\log n)$ times for s and then repeat for $O(\log n)$ samples of s itself.

$H_p(x)$ is monotonically non-increasing. The key idea behind defining H_p, as done originally in [113], is that H_p helps measure how fast the random walk is converging. If the walk stagnates, it would imply that one of the prefixes has a poor conductance.

With the assumption on the probability estimates and the above definitions, we have a crucial lemma bounding the difference in $H_p(x)$ and $H_{\tilde{p}}(x)$.

Lemma 5.5.4. *$|H_{\tilde{p}}(x) - H_p(x)| \leq O(\epsilon)$ with high probability. Also, $|H_{T_{\tilde{p}}}(x) - H_{T_p}(x)| \leq O(\epsilon)$.*

Proof. Notice that the set of nodes we are considering in $H_{\tilde{p}}(x)$ and $H_p(x)$ could be different. However, note that $|H_{\tilde{p}}(x) - H_p(x)| \leq |\tilde{p}_i - p|$.

Observe that $\sum_i p_i \leq 1$ and $\sum_i \sqrt{\frac{p_i}{n}} \leq 1$ and $\sum_i \frac{1}{n} \leq 1$ giving $\sum_i |\tilde{p}_i - p_i| \leq O(e)$. The other direction is obtained by looking at $1 - H_p(x)$. The bound on $|H_{T_{\tilde{p}}}(x) - H_{T_p}(x)|$ follows similarly. ∎

□

We begin by re-stating a lemma in [113](or Lemma 3.8 in [136]) in this context.

Lemma 5.5.5 ([113])**.** *If for all $j \in [1, n-1]$, $\Phi(\pi_p(\{1, \ldots, j\})) \geq \phi$, then*

$$H_{Tp}(x) \leq \frac{1}{2}(H_p(x - 2\phi\bar{x}) + H_p(x + 2\phi\bar{x})), \tag{7}$$

where $\bar{x} = min(x, 2m - x)$ *and* T *is the transition probability matrix*

This restatement follows from the fact that if p gives the probability distribution at length $l-1$, then Tp gives the probability distribution at length l. Since we use $\tilde{p}$, which is only an approximate distribution, we will need to modify as follows. Using the lemma above and Lemma 5.5.4, we have:

Corollary 5.5.6. *If for all* $j \in [1, n-1]$, $\Phi(\pi_{\tilde{p}}(\{1, \ldots, j\})) \geq \phi$, *then*

$$H_{Tp}(x) \leq \frac{1}{2}(H_p(x - 2\phi\bar{x}) + H_p(x + 2\phi\bar{x})) + O(\epsilon), \tag{8}$$

where $\bar{x} = min(x, 2m - x)$.

Proof. From Lemma 5.5.5, it follows that if for all $j \in [1, n-1]$, $\Phi(\pi_{\tilde{p}}(\{1, \ldots, j\})) \geq \phi$, then $H_{T_{\tilde{p}}}(x) \leq \frac{1}{2}(H_{\tilde{p}}(x - 2\phi\bar{x}) + H_{\tilde{p}}(x + 2\phi\bar{x}))$. Now the corollary follows from Lemma 5.5.4. ∎

□

We now state another lemma required subsequently.

Lemma 5.5.7 ([113])**.** *If* $F_{i+1}(x) \leq F_i(x)$ *and* $F_{i+1}(x) \leq \frac{1}{2}[F_i(x-2\phi\bar{x}) + F_i(x+2\phi\bar{x})]$ *where* $\bar{x} = \min\{x, 2m - x\}$, *and* $F_1(x) \leq \sqrt{\bar{x}}$, *then* $F_l < F_1(1 - \phi^2/2)^l + \epsilon l$.

These lemmas together gives us the corresponding result of Lemma 3.10 (Cut or Mix) in [136]. The basic idea is that either the distribution keeps mixing rapidly, or there is a cut across which it is not mixing. H measures how much the distribution is changing by. If the H keeps falling rapidly, that means that the initial distribution is quickly approaching the steady state distribution. If, however, it does not reach the steady state distribution fast, then there must have been a stage where H did not fall significantly in one step; observing the distribution at this stage (and ordering vertices in decreasing order of probabilities) gives n candidate cuts, of which at least one is *sparse*.

Lemma 5.5.8. *Either for half the values of $l \in [1, l_0]$, there exists a j such that $\Phi(\pi_{\tilde{p}_l}(\{1, \ldots, j\})) \leq \phi$, where $\tilde{p}_l$ denotes the estimate for the probability distribution after a walk of length l; Or $H_{l_0}(x) \leq \sqrt{\bar{x}}(1 - \phi^2/2)^{l_0/2} + \epsilon l_0/2$, for all $x \in [0, 2m]$.*

Proof. If for all j, $\Phi(\pi_{\tilde{p}_l}(\{1, \ldots, j\})) > \phi$, then $H_{l+1}(x) \leq \frac{1}{2}(H_l(x - 2\phi\bar{x}) + H_l(x + 2\phi\bar{x}))$. Further, even if this is not case, note that $H_{l+1} \leq H_l$. Now it follows from Lemma 5.5.7 that if for more than half the values of l, $\Phi(\pi_{\tilde{p}(l)}(\{1, \ldots, j\})) > \phi$, then $H_{l_0}(x) \leq \sqrt{\bar{x}}(1 - \phi^2/2)^{l_0/2} + \epsilon l_0/2$. ∎

□

Based on Corollary 5.5.8, we can show how to find a required cut (giving a theorem corresponding to Lemma 3.7(a) of [136]). Since we again have only approximate probability distributions, we cannot adopt their theorem directly.

Let S be the smaller side of cut of conductance Φ.

Definition 5.5.9. *For each set $S \subseteq V$, S^g is defined as the set of nodes $s \in S$ such that, the probability of ending up at $V \setminus S$ on performing a random walk of length l_0 from source s, is at most twice the probability of ending up at $V \setminus S$ on performing a random walk of length l_0 from the uniform source distribution over S.*

Proposition 5.5.10 ([136])**.** *$|S^g| \geq |S|/2$. Further, for any node in $s \in S^g$, the probability that a random walk of length l_0 ends up in $V \setminus S$ is at most $2\Phi(S)l_0$.*

This suggests that if the random choice of s is on the correct side of the cut, then the algorithm succeeds with probability at least 1/2. We now combine all these to get the following.

Theorem 5.5.11. *For a source $s \in S^g$ such that $Vol(S) \leq Vol(V)/2$, if we start a random walk at s of length l where l is chosen at random in the range of 1 to $l_0 = \frac{1}{8\Phi(S)}$, then, with probability at least $\frac{1}{2}$, there exists a $j \leq |S|$ such that $\Phi(\tilde{\pi}_t(\{1, \ldots, j\})) \leq \tilde{O}(\sqrt{\Phi(S)})$.*

Algorithm 20 AllNodesEstimation(s, l)

1: **Input:** Starting node/distribution s, length l.
2: **Output:** $\tilde{P}_l[sv]$ such that $|\tilde{P}_l[sv] - P_l[sv]| \leq \sqrt{\epsilon P_l[sv]/n} + \epsilon/n$ for each $v \in V$.
3: Perform $\Theta(n/d\epsilon)$ walks of length $l-1$ from s. Set $\tilde{P}_{l-1}[su]$ for each u to be the fraction of these walks with u as the end point.
4: For each v, set $\tilde{P}_l[sv] = \frac{1}{d}\sum_{(u,v)\in E} \tilde{P}_{l-1}[su]$.

Proof. We set $\phi = (\log n)\sqrt{\Phi(S)}$ in Lemma 5.5.8. This means that if we do not find a cut of conductance at most $\tilde{O}(\sqrt{\Phi})$, then $H_{l_0}(x) \leq \sqrt{\bar{x}}(1-\phi^2/2)^{l_0/2} + \epsilon l_0/2 \leq \sqrt{n}(1-\frac{\log n}{\Phi(S)})^{l_0/2} + \epsilon\frac{l_0}{2} \leq \sqrt{n}(1-\frac{\log n}{\Phi(S)})^{\Phi(S)/8} + \epsilon\frac{l_0}{2} \leq o(1) + \frac{\epsilon}{16\Phi(S)}$. We choose $\epsilon < \Phi(S)$ implying that $H_{l_0}(x) \leq o(1) + \frac{1}{16} \leq \frac{1}{15}$.

However, this contradicts Proposition 5.5.10 which states that the probability of ending up at $V \setminus S$ is at most $2\Phi(S)l_0 = \frac{1}{4}$; this is because this implies that $H_{l_0}(x) \geq \frac{1}{2} - \frac{1}{4} = \frac{1}{2}$.

This completes the proof. ∎

□

This theorem gives Theorem 5.2.2.

5.5.2 Finding sparse cuts on all nodes

We observe that our approach can be extended to finding sparse cuts on all nodes. In this section we will prove Theorem 5.1.8. We start by presenting Algorithm 20, AllNodesEstimation, that can be used to approximate the probability distribution of all nodes after a walk of length l, from source s for d-regular unweighted graphs. Unfortunately, it is not clear how successive RecursiveRandomWalk type techniques can be used efficiently if the entire vector of probability estimates is required. So, to estimate the probabilities after a walk of length l, we can do only slightly better than run n random walks of length l.

Suppose the probability of all nodes from source s at length l is required up to accuracy $p \pm \sqrt{p\epsilon/n}$ where $p = P_l[si]$. One can obtain this by performing n/ϵ walks of

length l and using the fraction of walks in which a node is the end point to estimate its probability. This, by the technique of [48] would require $\tilde{O}(n\sqrt{l}/\epsilon)$ space and $\tilde{O}(\sqrt{l})$ passes. We instead perform $\frac{n}{d\epsilon}$ walks of length $(l-1)$ from the source s. Then to obtain $P_l[sv]$, we take the average of the probabilities of neighbors of v from the estimate obtained by the n/d walks of length $l-1$. This estimation crucially uses the reversibility of the random walk process, which implies the graph is d-regular and unweighted.

Lemma 5.5.12. *For any subset of vertices U, $\tilde{P}_{l-1}[sU] = \sum_{u\in U} \tilde{P}_{l-1}[su]$ is within $P_{l-1}[sU] \pm \sqrt{\epsilon P_{l-1}[sU]d/n} \pm \epsilon/n$.*

Proof. In $\Theta(n/d\epsilon)$ walks of length $(l-1)$ from s, the expected number of walks that end in some node in U is exactly $P_{l-1}[sU]$. Our estimator of setting $\tilde{P}_{l-1}[su]$ for every u to the number of walks that end in u has the right expectation of $P_{l-1}[su]$. Therefore, clearly $E[\tilde{P}_{l-1}[sU]] = P_{l-1}[sU]$. Further, the error by standard Chernoff Bounds is $\sqrt{P_{l-1}[sU]/N} + 1/N$ where N is the number of samples drawn. Setting $N = n/d\epsilon$, the lemma follows as long as $P_{l-1}[sU]$ is large. For the case when this is small, we have a maximum error of ϵ/n. ∎

□

Once this lemma is established, it is easy to check that we get the desired accuracy for the probability of all nodes. We know that $P_l[sv] = \sum_u P_{l-1}[su].P_1[uv]$. But since the graph is d-regular, we have $P_l[sv] = \frac{1}{d}\sum_{(u,v)\in E} P_{l-1}[su]$. The lemma above gives, $\sum_{u,(u,v)\in E} |\tilde{P}_{l-1}[su] - P_{l-1}[su]| \leq \sqrt{\epsilon \sum_{u,(u,v)\in E} P_{l-1}[su]\epsilon d/n} + d\epsilon/n$. Since this error gets divided by d for $P_l[sv]$ the result follows immediately that AllNodesEstimation gives probability estimates for all nodes within an additive error of $p \pm \sqrt{p\epsilon/n} \pm \epsilon/n$.

These $\tilde{O}(\frac{n}{d\epsilon})$ random walks are performed on a graph stream, and so we use Theorem 5.2.3 and plug-in appropriate parameters to obtain the space and passes required.

Theorem 5.5.13. *One can estimate the probability of all nodes v from source s after a random walk of length l, $P_l[sv]$, within an additive error of $\sqrt{\epsilon P_l[sv]/n}+P_l[sv]/n$ by performing $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes over the graph stream and using space $\tilde{O}(\min\{n\alpha + \frac{nl\alpha}{d\epsilon} + \frac{n}{d\epsilon}\sqrt{\frac{l}{\alpha}}, (n\alpha + \frac{n}{d\epsilon})\sqrt{\frac{l}{\alpha}} + l\})$, for any $\alpha \leq 1$.*

Proof. In the space bound from Theorem 5.2.3, we have that k walks require, $\min\{\tilde{O}(n\alpha kl\alpha + k\sqrt{l/\alpha}), \tilde{O}(n\alpha\sqrt{l/\alpha} + k\sqrt{l/\alpha} + l)\}$ space. Choosing $k = \frac{n}{d\epsilon}$ completes the proof. ∎

□

Plugging in $l = O(1/\Phi)$ and $\epsilon = O(\Phi^2)$ as required by Theorem 5.2.2 immediately leads to Theorem 5.1.8 if we performed the above for $\frac{1}{b}$ sources. This is required because in Theorem 5.2.2, we need the source to be from the smaller side of the cut; and sampling which could take $\frac{\log n}{b}$ samples.

CHAPTER VI

A SKETCH-BASED DISTANCE ORACLE FOR WEB-SCALE GRAPHS

In this chapter, work from [47], we study the fundamental problem of computing distances between nodes in large graphs such as the web graph and social networks. Our objective is to be able to answer distance queries between a pair of nodes in real time. Since the standard shortest path algorithms are expensive, our approach moves the time-consuming shortest-path computation offline, and at query time only looks up precomputed values and performs simple and fast computations on these precomputed values. More specifically, during the offline phase we compute and store a small "sketch" for each node in the graph, and at query-time we look up the sketches of the source and destination nodes and perform a simple computation using these two sketches to estimate the distance.

6.1 Related Work and Contributions

Large graphs have become a common tool for representing real world data. We now routinely interact with search engines and social networking sites that make use of large web graphs and social networks behind the scenes. Many of these interactions happen in real time where users make some kind of a request or a query that needs to be serviced almost instantaneously. Since user interactions should have low latency, one method to handle expensive operations is to do them offline as a precomputation and store the data so that they can be obtained quickly when required for a real-time operation. For example, PageRank consists of an expensive eigenvector computation over the web graph that can be performed offline and a simple online lookup of the

PageRank score for each result.

A fundamental operation on large graphs is finding shortest paths between a pair of nodes. As we would like to answer distance queries in real time with minimal latency, it becomes important to use small amounts of resources per distance query even for massive graphs. In this work, we study the problem of estimating distances between two nodes in a large graph in real time, in an online manner.

A popular approach, that we also adopt in this work, is to perform a one-time offline computation. A straightforward brute force solution would be to compute the shortest paths between all pairs of nodes offline and to store the distances on disk. In this setting, answering a shortest-path query online requires a single disk lookup; however, the space requirement is quadratic in the number of nodes in the graph. For a web graph containing billions of nodes, this would be simply infeasible. To reduce the space complexity, our algorithms store some auxiliary information with each node that can facilitate a quick distance computation online in real time. This auxiliary information is then used in the online computation that is performed for every request or query. One can view this auxiliary information as a *sketch* of the neighborhood structure of a node that is stored with each node. Simply retrieving the sketches of the two nodes should be sufficient to estimate the distance between them. Two properties are crucial for this purpose: First, these sketches should be reasonably small in size so that they can be stored with each node and accessed for any node at run time. Second, there needs to be a simple algorithm that, given the sketches of two nodes, one can estimate the distance between them quickly. As the computation of the sketches is an offline computation, one can afford to spend more resources on this one-time preprocessing.

Computing sketches offline for a large collection of objects to facilitate online computations has been studied extensively and is also used in many applications. For example, several search engines compute a small sketch of documents to detect

near-duplicate documents at query time; see [34, 58, 33] and references therein. This eliminates the need to compare large documents, which is time consuming. Instead it is achieved by comparing short sketches of these documents and measuring the similarity between these sketches.

Computing sketches for the specific purpose of distance computation is also called distance labeling. Some papers that study the problem of the size of labels required with each node to allow distance computation include Gavoille et al. [70], Katz et al. [94], and Cohen et al. [41].

The field of metric embedding deals with mapping a set of points from a high-dimensional space to a low-dimensional space, such that the distortion is minimized. If each point can be projected onto a small number of dimensions such that distances are approximately preserved, one can store the small dimensional vector as a sketch. The classic result of Bourgain [30] shows how such an embedding can be achieved for certain distance metrics.

Another line of work in estimating distances is the study of *spanner* construction. A spanner is a sparse subgraph of the given graph, such that the distance on this sparse graph approximates the actual distance, for any pair of points. Although spanners take small space, they do not exactly provide a sketch for each node; thus the online algorithm for estimating distance may take a long time. Some theoretically efficient algorithms for spanners are presented by Feigenbaum et al. [65], and Baswana [20]. Other fundamental results in this area include Bartal [19] and Fakcharoenphol et al. [62].

Cohen et al. [40] proposed an approximate distance scheme using 2-hop covers of all paths in a directed (or undirected) graphs. However, finding the near optimal 2-hop cover of a given set of paths is expensive in a large graph. Moreover, the size of such a cover can be as large as $\Omega(nm^{1/2})$ make their scheme quite hard to implement on large graphs.

Several studies, for example the ones by Goldberg et al. [73, 54], have focused on answering exact shortest path queries on road networks. These algorithms make use of a small set of precomputed landmarks and shortcuts and use them at query time to connect a source, destination pair; these landmarks and shortcuts are chosen very carefully using algorithms that are specialized to the structure of road networks. Our algorithms can be viewed as using a randomly sampled set of landmarks; as such, it can be taken to mean that even a simple algorithm that make use of randomly sampled 'landmarks' work very well on othe complex graphs such as the web graph.

6.1.1 Our Contributions

In this work, we engineer algorithms for computing such sketches and demonstrate how they can be used to estimate distances between arbitrary pair of nodes, in real time. While there is not much empirical studies on computing sketches for distances, there are the aforementioned theoretical studies using embeddings [30, 19, 62] and spanners [65, 137, 20] in the algorithms literature. All of these algorithms provably work only on undirected graphs and some are complicated and probably impractical. The classical result by Bourgain [30] shows how one can project a graph onto a low dimensional space giving a small sketch that approximates distances to a factor of $O(\log n)$; Matousek [119] later showed that the same algorithm can be used to get a $2c-1$ factor approximation using sketches of size $\tilde{O}(n^{1/c})$. However we find in our experiments that Bourgain's algorithm performs very poorly in estimating the distance. On the other hand, we propose an algorithm that is essentially a simplification of the algorithm by Thorup and Zwick [137] and provides the same theoretical guarantee: our algorithm approximates distances within a factor $2c-1$ by using sketches of size $\tilde{O}(n^{1/c})$ (note that for $c = \log n$ this gives a $O(\log n)$-factor approximation to the distances using sketches of size $O(\log n)$). Furthermore, our algorithm is simpler to implement. While both algorithms sample seed sets of different

sizes and find the closest seed in each seed set, the Thorup and Zwick algorithm needs to store additional data in the sketch of a node – they store all ids of nodes in a seed set that are closer than the nearest seed in the next seed set when ordered by decreasing size; this adds more complexity to the offline precomputation and also introduces some additional checks beyond what we do in the online step. In this work, we are demonstrating that the additional data in the sketch is not required, and simply doing the keeping the seed id along with the distance is sufficient.

Also, unlike the previous works, we extend our algorithms to directed graphs and evaluate them on large graphs. We find in our experiments that even on a large web graph, our algorithm gives very good estimates for both directed and undirected distances – the distances are usually accurate to within an additive error of 2 to 3. The fact that our algorithm performs well for directed distances is surprising as there is no known sketching algorithm for directed distances; in fact, it is known that this is impossible in the worst case for arbitrary directed graphs, which suggests that the web graph has some special properties. Understanding the gap between the theoretical guarantee and the experimental observation is an interesting area for future investigation.

The essential idea behind our algorithm is the following: In the offline computation, sample a small number of sets of nodes in the graph (sets of seed nodes). Then, for each node in the graph, find the closest seed in each of these seed sets. The sketch for a node simply consists of the closest seeds, and the distance to these closest seeds. Then, in the online computation, one can use the distance to this closest seed to estimate the distance between pair of nodes. One method to do this is to check if there is a common node between the two sketches. Given a pair of nodes u and v one can estimate the distance between them by looking for a common seed in their sketches. So if w is a common seed in the sketch of u and v then the distance can be estimated by adding up the distances to the common seed w. We also note

that our algorithm always produces an upper bound on the actual distance; whereas Bourgain's algorithm always produces a lower bound.

Experimentally we observed that our algorithm performs very well for estimating both directed and undirected graphs. We perform experiments on a 2002 crawl of the web graph containing about 419 million URLs and 2.37 billion edges. We compare our algorithms with actual distances computed by Dijkstra's algorithm between sampled pairs of nodes. The actual undirected distances are in the range of 1 and 15, and the directed distances (whenever connected by a directed path) are in the range of 1 and 100. We sample pairs from each of these distances for examining the performance. For all pairs u, v with actual distance $d(u, v)$, we compute the median of the distance estimate obtained by each of our algorithms. We do this independently for undirected and directed distances.

6.2 Our Algorithms

We will now describe the algorithm to compute sketches for each node that can be used to perform online distance computations between a pair of nodes u and v. For simplicity, we only work with undirected graphs in this section, and generalize to directed graphs in Section 6.3.

We denote a graph by G, its nodes by V, edges by E, and number of nodes $|V|$ by n. The distance between u and v is denoted by $d(u, v)$. Our goal is to preprocess and store this graph in such a way that given any pair of nodes u and v, we are able to estimate the distance $d(u, v)$ in real time using a small amount of computation. It is important to keep in mind that a web-scale graph contains tens of billions of nodes and trillions of edges. The precomputation consists of computing a sketch $\textsc{Sketch}[u]$ for each node u. The real time computation of the distance between u and v should involve only reading $\textsc{Sketch}[u]$ and $\textsc{Sketch}[v]$.

We will use $\tilde{d}(u, v)$ to denote an estimate for $d(u, v)$ obtained by a sketching

Algorithm 21 OFFLINE-SAMPLE(G)

Input: An undirected graph G and a node u in the graph.
Output: SAMPLE[u], A set of nearest seeds and their distances for each node u.

1: Let V denote the set of vertices, let $r = \lfloor \log |V| \rfloor$. Sample $r+1$ sets of sizes $1, 2, 2^2, 2^3, ..., 2^r$ respectively. In each set, the samples are chosen uniformly at random from all nodes. (As stated before, for the sampling, we may ignore from the set V, nodes with in-degree or out-degree zero as they cannot be in the shortest path between a source and a destination). Call the sampled sets $S_0, S_1, \ldots, S_r$.
2: For each node u, for all these sets S_i, compute (w_i, δ_i) where w_i is the closest node to u in S_i and $\delta_i = d(u, w_i) = d(u, S_i)$.
This can be done for all nodes u efficiently with just one BFS from each set S_i.
SKETCH[u] $= \{(w_0, \delta_0), \ldots, (w_r, \delta_r)\}$

algorithm. Another notation that we use in this chapter is $d(u, S) = \min_{w \in S} d(u, w)$, which is the shortest distance between u and a set of nodes $S \subseteq V$.

6.2.1 Algorithm Offline-Sketch

As stated earlier, the essential idea in our algorithm is to sample a set of seed nodes, and store the closest seed from every node along with its distance. This can be done efficiently by just one Breadth First Search (BFS) from the seed set (note that it is even possible to find the nearest node in the set from each node by passing node IDs appropriately while performing the BFS). The nearest node in a sampled set from a node u is referred to as the seed of u in that set. We do this for $\log n$ sets of sizes that are different powers of 2 in function OFFLINE-SAMPLE (see Algorithm 21) which returns the nearest seed and the distance to it in each of these sets. Function OFFLINE-SKETCH (see Algorithm 22) repeats this k times – each time using different random sets – and returns the union of the samples. All of this is done offline and stored as SKETCH[u] for each node u in the graph.

In order to help in estimating the distance between a source vertex u and destination vertex v, the seed vertex must lie on a path between u and v. In order to lie on a directed path, a vertex must have non-zero in- and out-degree. Thus, we should

Algorithm 22 OFFLINE-SKETCH(G)

Input: An undirected graph G and a node u in the graph.
Output: SKETCH[u], the sketch of node u.

1: Run OFFLINE-SAMPLE(G) k times independently with different random numbers each time
2: SKETCH[u] = Union of the SAMPLE[u] returned by each of the k runs.

consider only such vertices as candidates for a seed set. In the undirected case, we should consider only vertices with degree of at least 2.

The above candidate selection rule applies to any graph. There are other rules that may improve the accuracy of our algorithms further (for example, biasing the sampling process towards high-degree nodes), but they are dependent on the topology of the graph (for example, biasing toward high-degree nodes is a bad strategy in a "dumbbell" graph); therefore, we did not consider them in this work.

6.2.2 Algorithm Online-Common-Seed

We now present our main algorithm that estimates distances using the sketches. Given nodes u and v, algorithm ONLINE-COMMON-SEED (see Algorithm 23) approximates the distance between them by looking at the distance from u and v to any node w occurring in bot SKETCH[u] and SKETCH[v]. The length of the shortest path from u to v through w is $d(u,w) + d(w,v)$; note that both $d(u,w)$ and $d(w,v)$ are contained in the sketches and do not need to be computed. We take the minimum of $d(u,w) + d(w,v)$ over all such common seeds w to be the estimated distance $\tilde{d}(u,v)$. It is easily shown that this distance estimated by ONLINE-COMMON-SEED is always an upper bound on the actual distance. If the path with the corresponding length is desired, then the next hop on the paths to the seed can be stored during the sketch computation phase.

OBSERVATION. The estimated distance $\tilde{d}(u,v)$ from ONLINE-COMMON-SEED(u,v)

Algorithm 23 ONLINE-COMMON-SEED(u,v)

Input: Nodes u and v from G between which the distance is to be estimated.
Output: An estimate of $d(u, v)$.

1: Obtain SKETCH[u] and SKETCH[v].
2: Find the set of common nodes w in SKETCH[u] and SKETCH[v]. Note that there is at least one w for undirected G assuming that G is connected, as we perform a BFS from at least one set of size 1.
3: For each common node w, compute $d(u, w)$ and $d(w, v)$.
4: Return the minimum of $d(u, w) + d(w, v)$, taken over all such common w's. If no common seed w is present, then output ∞.

is an upper bound of the actual distance $d(u, v)$.

Proof. The algorithm considers various nodes w that are in the intersection of the two sketches of u and v. From the sketches obtained offline, we can compute exactly, $d(u, w)$ and $d(w, v)$ for these nodes w. By triangle inequality, $d(u, w) + d(w, v) \geq d(u, v)$. Notice that we take the minimum sum over several w, but for each of these nodes, the triangle inequality holds. The observation follows. □

Theorem 6.2.1. *The estimated distance $\tilde{d}(u, v)$ returned by* ONLINE-COMMON-SEED(u,v) *for $k = \tilde{\Theta}(n^{1/c})$ in the sketch computation gives with high probability*[1] *a $2c - 1$ approximation to the actual distance for all pairs; $d(u, v) \leq \tilde{d}(u, v) \leq (2c - 1)d(u, v)$.*

Proof. Let $d = d(u, v)$. Let A_r, B_r denote balls of radius rd around u and v respectively; that is, all nodes within distance at most rd around u and v.

Consider the points in $A_r \cup B_r$ and $A_r \cap B_r$. If one of the seed sets S is such that it has exactly one seed w in the union $A_r \cup B_r$ which is also in the intersection $A_r \cap B_r$, then w is a common seed in the sketch of u and v. This is because it is the closest seed to both u and v. We will argue that such a seed set exists and results in a common seed in the sketch that is at distance at most $d \log n$ from both u and v.

[1] *with high probability means with probability* $1 - 1/n^{\Omega(1)}$

For simplicity, let us consider the case when $c = \log n$. Precisely, we observe that if $\frac{|A_r \cap B_r|}{|A_r \cup B_r|}$ is at least some constant (say $1/2$), then when we take seeds with probability $\frac{1}{|A_r \cup B_r|}$, there is a constant chance that exactly one seed is present in the union which also happens to be in the intersection. If seeds are sampled with probability $\frac{1}{|A_r \cup B_r|}$, this event happens with probability at least $1/(2e)$ since with probability $1/e$ there is exactly one seed and further with probability $1/2$ it lies in the intersection. Since we are trying seed set sizes that are different powers of 2, the probability will be constant for the closest power of 2. Thus, if $\frac{|A_r \cap B_r|}{|A_r \cup B_r|} > 1/2$ for any i in the range $1..\log n$, then there is a constant probability of finding a common seed within distance $d \log n$ from both u and v. If not, this means $\frac{|A_r \cap B_r|}{|A_r \cup B_r|} \leq 1/2$ for all $1 \leq i \leq \log n$.

But note that $A_r \cup B_r \subseteq A_{r+1} \cap B_{r+1}$ since the set on the left hand side contains points at distance at most rd from u or v, and further since $d(u, v) = d$, these points are at distance at most $(r+1)d$ from both u and v implying that they are all present in $A_{r+1} \cap B_{r+1}$. So if $\frac{|A_r \cap B_r|}{|A_r \cup B_r|} \leq 1/2$ for all $1 \leq i \leq \log n$, this means $|A_{r+1} \cup B_{r+1}| > 2|A_r \cup B_r|$, implying $|A_{\log n} \cup B_{\log n}| > n$ which is impossible. Thus there must be a value of r such that $\frac{|A_r \cap B_r|}{|A_r \cup B_r|} \leq 1/2$. Since we try each size k times, for constant k, we can make the probability of failure negligible. The same proof generalizes to arbitrary c; we show that is some i, $1 \leq r \leq c$, such that $\frac{|A_r \cap B_r|}{|A_r \cup B_r|} \geq n^{-1/c}$; if we repeat $k = \tilde{\Theta}(n^{1/c})$ times, we succeed with high probability, giving a $2c$ approximation. This can be strengthened to give a $2c - 1$ approximation by looking at the sets $A_r \cup B_{i-1}$ and $A_r \cap B_{r-1}$ – note that $|A_r \cap B_{r-1}| = 1$ and for any point in the intersection the sum of the distances from u and v is at most $i \cdot d + (r-1)d = (2r-1)d \leq (2c-1)d$. □

6.2.3 Algorithm Online-Bourgain

We compare the performance of our algorithm to that of Bourgain's well-known technique that embeds some metric space into a low dimensional space. We describe this in Online-Bourgain (see Algorithm 24). We note that the same sketches are used

Algorithm 24 ONLINE-BOURGAIN(u,v)

Input: Nodes u and v from G between which the distance is to be estimated.
Output: An estimate of $d(u, v)$.

1: Obtain SKETCH[u] and SKETCH[v].
2: For each seed set S, extract $d(u, S)$ from SKETCH[u] and $d(v, S)$ from SKETCH[v], and compute $|d(u, S) - d(v, S)|$.
3: Return the maximum $|d(u, S) - d(v, S)|$ over all seed sets S.

as before. However, instead of finding nodes in the intersection of both sketches, we find the distance from v to all the seed sets and similarly the distance from u to all the seed sets. The L_∞-norm of the difference of these two vectors gives a lower bound on the actual distance between u and v.

Furthermore, Bourgain also proves that if we set k to $\Theta(\log n)$ (where k is the number of sets picked for each of the $O(\log n)$ sizes), then the distances are accurate up to constant factor of $\log n$. By using seed sets of size powers of 2 and using k sets of each size, one can prove theoretical guarantees on the quality of estimates. We present a slight alteration that can be extended to directed graphs. While the intuition carries over to directed graphs, unfortunately one can no longer prove a $O(\log n)$ approximation ratio. However, we present the simple observation that the returned distance estimate $\tilde{d}(u, v)$ is a lower bound on the actual distance $d(u, v)$. Notice that in the algorithm, we consider the absolute value of $d(u, S) - d(v, S)$ for undirected graphs. This is justified by showing in our proof that both directions impose a lower bound, and hence the absolute value does too.

OBSERVATION. The distance estimate $\tilde{d}(u, v)$ returned by ONLINE-BOURGAIN(u,v) is a lower bound on the actual distance $d(u, v)$ from u to v.

Proof. The proof follows essentially from triangle inequality. We will show that for any set S of nodes $|d(u, S) - d(v, S)| \leq d(u, v)$. It is sufficient to show that $d(u, S) -$

$d(v,S) \leq d(u,v)$ (as by symmetry this will also imply that $d(v,S) - d(u,S) \leq d(u,v)$). To see this, let $d(v,S) = d(v,v')$ where v' is the closest node to v in S. $d(u,v') \leq d(u,v) + d(v,v')$ by triangle inequality, which gives $d(u,v') \leq d(u,v) + d(v,S)$. So $d(u,v) \geq d(u,v') - d(v,S) \geq d(u,S) - d(v,S)$ since u' is the closest node to u in S. Now the estimate $\tilde{d}(u,v)$ is obtained by taking the maximum of $|d(u,S) - d(v,S)|$ over different sets S used in the offline phase. But since for each S, the difference is bounded by $d(u,v)$, the maximum is also bounded by $d(u,v)$ □

Matousek [119] proved that Bourgain's algorithm ONLINE-BOURGAIN gives a $2c-1$-approximation to distances using sketches of size $\tilde{O}(n^{1/c})$ However, note that this result holds only for undirected graphs.

Theorem 6.2.2. *The distance estimate $\tilde{d}(u,v)$ computed by* ONLINE-BOURGAIN(u,v) *for $k = \Theta(n^{1/c})$ in the sketch computation is with high probability a $O(2c-1)$ factor approximation; $\Omega(\frac{d(u,v)}{2c-1}) \leq \tilde{d}(u,v) \leq d(u,v)$.*

Proof. Given in [30] □

6.3 Generalization to Directed Graphs

The algorithms presented in the previous section can easily be extended to the directed case. However, there are no known theoretical guarantees except for the upper and lower bounds we have shown. We state the main differences in the directed algorithms as compared to the undirected algorithms by rewriting the changed subroutines.

6.3.1 Modification to Online-Common-Seed

As before, the accuracy of the upper bound improves with the number of sets sampled in the offline sketch computing phase. It is important to choose sets of different sizes (to capture the right distance) as well as multiple sets of each size (to reduce the sensitivity of finding exactly the same nearest node from both u and v).

In the directed version of Offline-Sample, we compute both direction distances, $d(u, S)$ and $d(S, u)$. For each sampled S, we find w_1 in S so that $d(u, w_1) = d(u, S)$ (i.e. $d(u, w_1)$ is minimized over $w_1 \in S$), and w_2 such that $d(w_2, u) = d(S, u)$ (i.e. $d(w_2, u)$ is minimized over $w_2 \in S$). One can obtain this efficiently for all nodes u with just two BFS runs from S. One run starts with S and iterates by traversing along incoming edges. The other run uses the outgoing edges.

The online distance computation algorithm then considers all w in the sketches of u and v. However, to compute the directed distance $d(u, v)$, we consider all w in the sketch corresponding to the distance from u to S and for v, the sketch corresponding to the distance from S to v. The distance estimate is then computed as before, $\tilde{d}(u, v) = d(u, S) + d(S, v)$.

The proof of the upper bound for the directed case follows in the same way as the undirected case, using triangle inequality for directed distances.

6.3.2 Modification to Online-Bourgain

The algorithm for finding the lower bound estimate of $d(u, v)$ in directed graphs is a minor modification of the algorithm for undirected graphs. We describe it in Online-Bourgain-Directed(u,v) (see Algorithm 25).

One difference is that when we consider $d(u, S) - d(v, S)$, we use the maximum between this and 0 and not the absolute value. This is crucial since directed graph distances do not satisfy the symmetry property of a metric. We did not have to consider this in the undirected algorithm as there always was a path between every node and every S and distances are symmetric. However, in directed graph, there may not be a path from/to a node to/from the set (as the input graph need not be strongly connected). Therefore, the quantity we are subtracting may turn out to be ∞ (so taking max eliminates obtaining negative values). Notice that we do not need to worry about the distance becoming positive ∞ (this can be verified from

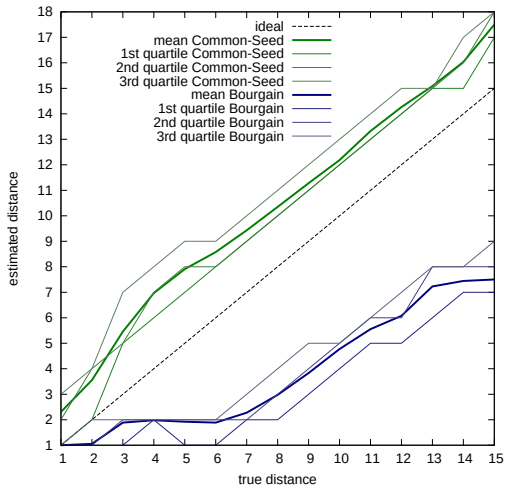


Figure 6: Estimates of undirected distances with $k = 1$

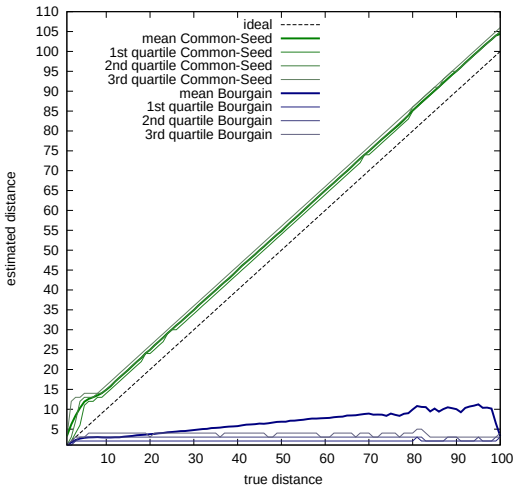


Figure 7: Estimates of directed distances using with $k = 1$

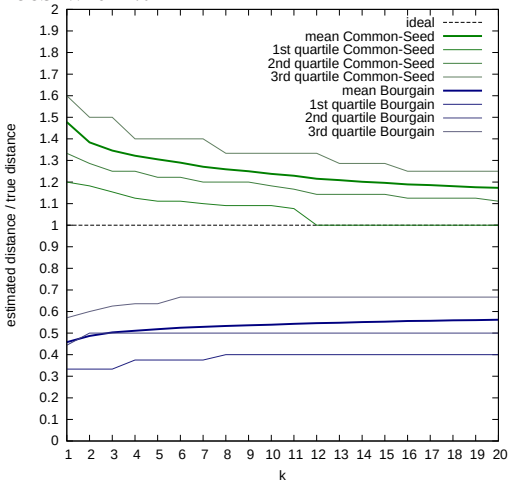


Figure 8: Estimates of undirected distances using as a function of k

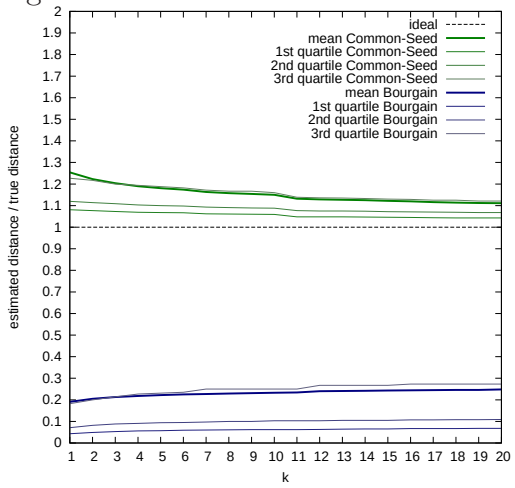


Figure 9: Estimates of directed distances using as a function of k

the correctness of the lower bound observation). Another minor difference is that we compute two quantities here, $d(S, v) - d(S, u)$ as well as $d(u, S) - d(v, S)$. This only gives us a stronger lower bound, as we shall show that both quantities independently are lower bounds.

OBSERVATION. The distance estimate $\tilde{d}(u, v)$ returned by ONLINE-BOURGAIN-DIRECTED(u,v) is a lower bound on the actual distance $d(u, v)$ from u to v.

Algorithm 25 ONLINE-BOURGAIN-DIRECTED(u,v)

Input: Nodes u and v from G between which the distance is to be estimated.
Output: An estimate of $d(u, v)$.

1: Obtain SKETCH[u] and SKETCH[v].
2: For each set distance in sketch corresponding to a set S, extract $d(u, S)$, $d(S, u)$ and $d(v, S)$, $d(S, v)$.
3: Compute $\max\{0, d(S, v) - d(S, u), d(u, S) - d(v, S)\}$ and find the maximum of this quantity over distances corresponding to all sets used to compute sketch from outgoing edge and incoming edge BFSs.
4: Maximum of the above two steps is returned as $\tilde{d}(u, v)$.

Proof. We need to show that the triangle inequality holds for both steps. Since we are taking the max with 0, it is sufficient to show that both $d(S, v) - d(S, u) \leq d(u, v)$ and $d(u, S) - d(v, S) \leq d(u, v)$. It is important to maintain the directed distances here. Rearranging, these follow from triangle inequality, using the sequence of arguments described in Observation 6.2.3, for every set S. Since we are taking the maximum over certain sets to compute $\tilde{d}(u, v)$, this is a lower bound of $d(u, v)$. □

In the following section, we compare the performance of the algorithms on large web graphs.

6.4 Experiments

Our experiments are performed on a large crawl of the web graph. We present some basic statistics here. The crawl was conducted in 2002 and is the result of a breadth-first search crawl starting at `www.yahoo.com`. We conducted our experiments on prefixes of the crawl. The number of crawled web pages is 65,581,675 and the number of distinct URLs is 419,545,168. This means that there are about five times as many nodes in the uncrawled "frontier" as in the explored part of the graph. The total number of edges in the graph is 2,371,215,893. The average out-degree of crawled pages is 36.16, and the average in-degree of all pages (whether crawled or not) is 5.65. The maximum out-degree and in-degree are 27,764 and 1,402,576 respectively.

In order to evaluate our algorithm on a given graph, we sampled 100 nodes at random. For each node v, we computed the distance between v node and all other nodes u. In the directed case, we computed the distance from all nodes u to v (which is ∞ if there is no path from u to v). We grouped the u's by distance and selected (up to) 10 nodes at random from each group, i.e. for each given distance. This left us with a test set of (u, v, d) triples for each given graph.

In order to conduct our experiments, we used the Scalable Hyperlink Store [127], a distributed system that partitions the web graph across many SHS servers, with each server maintaining a portion of the graph in main memory. Our implementation consists of an offline phase and an online phase. During the offline phase, we select the $\log n$ seed sets of exponentially increasing sizes and compute distances between every node in the graph and its closest seed using Dijkstra's algorithm. For each seed set, we output a temporary file consisting of a `seedid, distance` pair for each vertex in the graph. At the conclusion of the offline phase, we merge all the temporary files into a single sketch file consisting of $\log n$ pairs for each vertex. In directed graphs, we run the offline phase twice – once to compute distances from seeds to vertices, and once from vertices to seeds. To implement Online-Common-Seed, we repeat the offline phase k times. Finally, we merge the k (or $2k$) files into a single file.

During the online phase, we read the `seedid, distance` vectors of both the source and the destination vertex from disk. So, each query involves two disk seeks. Given that a disk seek takes several milliseconds while the subsequent processing of the sketches takes only microseconds, our algorithm is as fast in practice as Thorup's and Zwick's algorithm, despite the fact that from a theoretical complexity perspective, their algorithm has better time bounds for the online computation, requiring $O(c)$ time for a $(2c-1)$-approximation while ours requires $\tilde{O}(n^{1/c})$.

6.4.1 Single seed set sampling (k=1)

In our data set, true pairwise undirected distances vary between 1 and 15. We sample pairs to cover most of these distances and then query their sketches to find approximate distances. The directed graph is not strongly connected. But for pairs where there is a directed path, the distance spans the range of 1 to 100.

Figure 6 plots the estimates obtained from the algorithms Online-Common-Seed and Online-Bourgain respectively. In both plots, the x-axis denotes the actual distance of the sampled pair, the y-axis denotes the estimated distance. Therefore, the $x = y$ line corresponds to a perfect prediction of the actual distance.

We show curves for the mean, median, 75th percentile and 25th percentile. The same plot for directed distances is shown in Figure 7. We note that only one set of each of the sizes has been sampled to compute the sketch (i.e., $k = 1$). Recall that the distance returned by Online-Common-Seed is based on finding common nodes in the sketches of the two query nodes. On the other hand, the lower bound algorithm Online-Bourgain is based on computing the L_∞ distance between distances of query nodes to sampled sets.

Consider the plots for undirected distances first. Notice in Figure 6 that all the three quartiles produced by Online-Common-Seed are fairly close to $x = y$. In fact, even around the maximum true distance of 15, we obtain an upper bound of about 18 even at the 75th percentile. This suggests a 1.2-approximation ratio. Considering that all the previous approaches can only guarantee a $O(\log n)$-factor approximation (sometimes with large constants) and do not scale well, this is a very good approximation. In the same figure, we see that the lower bound produced by Online-Bourgain is also reasonably good (steadily increasing). However, it is noisier than the upper bound, and with a weaker guarantee, of about 15/7 which is 2.14-approximate.

In the case of directed distances (again with just one set sampled for each of

the $\log n$ sizes), Figure 7 illustrates a large gap between the estimates produced by Online-Common-Seed and Online-Bourgain. While Online-Common-Seed produces an extremely good 1.05-approximate estimate of the true distance, Online-Bourgain performs rather poorly. This is due to the fact that the number of sampled sets was not enough to capture a directed path between many pairs of nodes. Indeed, it is very difficult to capture a directed path through a sampled seed set when there may be very few directed paths. In the following set of experiments, we study the effect of larger sketches on the performance of both algorithms.

6.4.2 Using larger sketches to improve the bounds

We can improve the large gap in the directed distance estimation by additional sampling. We construct $k = 20$ sets of each of the $\log n$ different sizes, and use them to construct the sketch of every node. The hope is to capture many more directed paths between query nodes and the sets (which are available in the sketch) since the total number of sets in the sketch has gone up from $\Theta(\log n)$ to $\Theta(k \log n)$. This in turn yields better estimates of the directed distance between the two query nodes. The bounds computed using a larger number of sampled sets are plotted in Figure 8 and 9. Instead of plotting the actual estimated distance, we plot the ratio of estimated distance to true distance on the y-axis. Thus, this value will be above the ideal $y = 1$ line for the upper bound and below this line for the corresponding lower bound.

The sampling turns out to dramatically improve the upper bounds. All three – median, 75th percentile, and 25th percentile approach $y = 1$ as the number of samples increase. Note that the upper bound estimate reduces from a 1.6-approximate value for $k = 1$ to a 1.25-approximate value for $k = 20$. This shows that the algorithm Online-Common-Seed is successful in computing directed distance estimates to a high accuracy as well, provided sufficient sampling is done in the offline sketching phase. On the other hand, the lower bound from Bourgain's algorithm still does not

quite match the performance of Online-Common-Seed though it does produce a 1.85-approximate distance value for $k = 20$.

In the above experiment, we compute the quartile values for the ratios over all distances. To visualize the effectiveness of our algorithms for any given distance, we plot the median value of the ratio of the estimated distance to the actual distance for the undirected and directed cases in Figures 10 and 11 respectively. Further, we consider three values of k, 1, 10, and 20, to also illustrate the effect of the number of samples. As these figures show, the accuracy of the upper bound increases as the true distance increases. This is explained by the low likelihood of two vertices having a common seed in one of the smaller seed sets. This trend is not observed for the lower bound where, in fact, the ratio for the lower bound approaches a value close to 0 for large distances. The second observation is that the number of samples also helps in improving the distance estimates in both directions, albeit more in the case of the upper bound. We note that a sample size of $k = 10$ gives a good approximation for true distances above 10. This suggests that we can achieve good performance from Online-Common-Seed using a reasonably small number of samples.

To illustrate the spread of the error in distance estimation, we compute the distribution of the estimated distances for a given distance. Figures 12 and 13 show the distribution of the error for the lower and upper bounds where the distance $d = 10$ is chosen for the undirected case and distance $d = 50$ is chosen for the directed case. We chose these values of distance from the middle of the range for each case. Again, we run the experiment for three values of k: 1, 10, and 20. As we can see from the figures, there is a sharp concentration of around 0.5 (2-approximate) for the lower bound that does not change much with k. Further, the value of k does not affect the the sharpness of the concentration. On the other hand, for values produced by Online-Common-Seed, we observe two trends. First, the sharpness of the concentration increases with k. Second, the concentration shifts toward the ideal value of 1

as k increases.

In the case of directed distances, Online-Common-Seed produces a much better distribution with a concentration around 1.05 while Online-Bourgain produces a concentration close to 0.1. We see the concentrations shift slightly toward the ideal value of 1.0 (from both directions) as the value k increases showing the effectiveness of using more number of samples.

Note that in the earlier experiments, we ignore all pairs between which our algorithm could not find a path. To show that there are not many such entries, we compute the fraction of sampled pairs for which the algorithms fail to estimate a finite distance. This number is computed for different values of k. Note that for undirected distances, there are no such pairs as there is always a path between any two nodes in the graphs we consider. Therefore, we present the results only for the undirected case. Figure 14 illustrates the trend in the fraction of uncovered pairs as the number of samples increases. This value quickly drops to 0 even as k increases slightly above 2. This trend is observed for almost all the graphs. This result combined with that in Figure 9 shows that our algorithm finds a good approximation to the actual directed distance between *any* given pair of vertices.

6.4.3 Effect of graph size

In another set of experiments, we varied the graph sizes by choosing different prefixes of the breadth-first-search crawl and estimated both directed and undirected distances for a sampled set of vertex pairs. Specifically, we considered graphs of sizes between 47 million and 419 million nodes. The corresponding number of edges varied from about 171 million to 2.3 billion. Figures 15 and 16 illustrates the effect of graph size on the estimation of undirected and directed distances respectively. While there is no noticeable effect on the estimated distance in the undirected case, Online-Common-Seed tends to do better with larger graph sizes. One explanation for this

phenomenon could be that there are likely to be more paths between nodes in a larger graph and hence can be captured by the seed sets.

6.4.4 Effect of seed size

So far we have used 32-bit unique node identifiers to represent the seeds in our sketch computation. We will now explore the effect of lossy compression of the seeds. We hash the 32-bit seed representation to a representation with fewer, say b, bits. Note that the seeds are not used in Online-Bourgain and hence the performance of this algorithm is not affected. In the algorithm Online-Common-Seed, we use the seeds in the sketch to identify the common seed in the sketches of a given pair of nodes. Hashing the seed id into a representation with fewer bits might produce false positives of common seeds in the process. In fact, the probability of two seeds hashing to the same value is $1/2^b$ and since the number of distinct pairs of seeds across the two sketches grows quadratically in k, the total probability of a false positive is proportional to $\frac{k^2}{2^b}$. Note that because of such false positives, the algorithm may not necessarily return an upper bound of the true distance. Not surprisingly, we observe that for small values of k and b, we obtain reasonable accuracy in the estimates of the upper bound, i.e., we see a tiny fraction of false positives. Figures 17 and 18 show the fractional mismatches for different values of b and k. In the undirected case, we need fewer bits to achieve the same accuracy compared to the directed case. However, as k increases, the number of bits required to reduce the false positives also increases as the probability of a false positive is proportional to k^2. We observe that for $k = 3$ and 12 bits per seed, the fraction of mismatches is almost negligible. Further, the error in the estimated distance for $k = 3$ is also small (see Figures 8 and 9). This sketch size (in bits) can be computed as $(s + 8)k \log n$, where the $\log n$ factor comes from the number of seed sets and s is the number of bits per seed. The additional 8 bits store the distance. Setting $k = 3$, $s = 12$ and the number of seed sets to 32, we get a

sketch size of 240 bytes for undirected graphs and 480 bytes for directed graphs.

6.5 Summary

We present an algorithm for obtaining sketches that support efficient distance queries. The sketches are computed offline and distances between any pair of nodes can be estimated online by looking at their sketches. While there has been theoretical work for this problem, few of the approaches scale to web graphs. We present an algorithm that can approximate true distances and provide strong theoretical guarantees for the upper bound achieved by our algorithm Online-Common-Seed. While all previous algorithms have been suggested for distances on undirected graphs, we mention how our techniques can be extended to compute directed distances as well. Further, we compare our algorithm with Bourgain's well-known algorithm based on embedding graphs into low-dimensional metric spaces. The lower bound given by Bourgain's algorithm turns out to be significantly weaker. It is unable to predict directed results with any precision. For undirected graphs, while it is unable to match our upper-bound algorithm in terms of accuracy, it does give a reasonably good prediction.

We conduct extensive experiments of all the algorithms proposed in this work and compare it with true distances for sampled pairs of nodes. The experiments are run on a crawl of a large web graph As far as we are aware, this is the first practical work on distance oracles that has been adapted to real data at this scale.

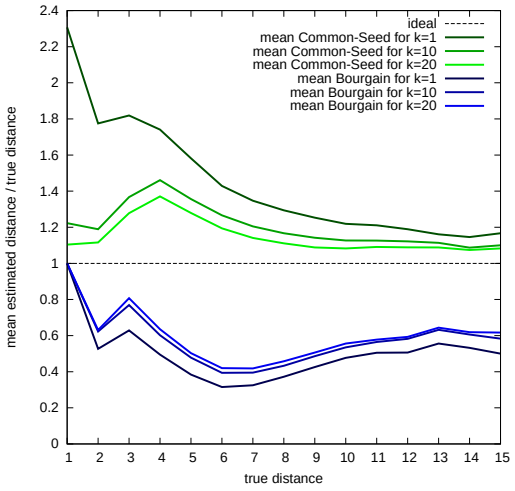


Figure 10: Ratio of estimated distance to true distance for different values of undirected true distance

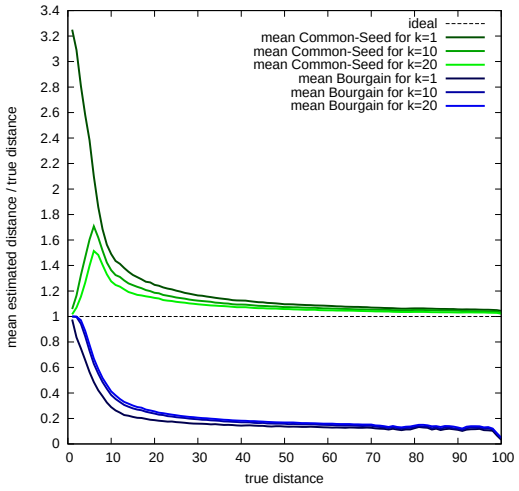


Figure 11: Ratio of estimated distance to true distance for different values of directed true distance

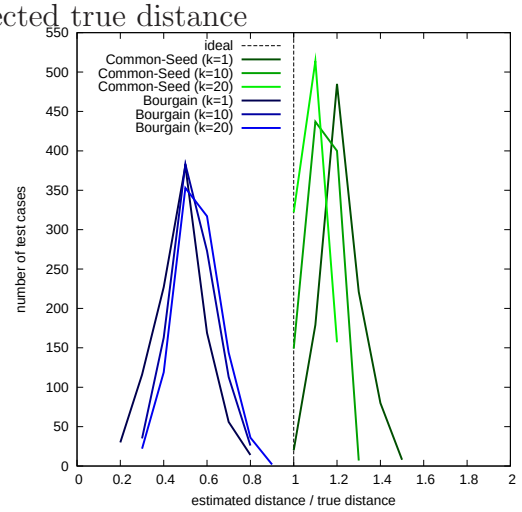


Figure 12: Distributions of the ratio of estimated distance to true distance for undirected distance $d = 10$

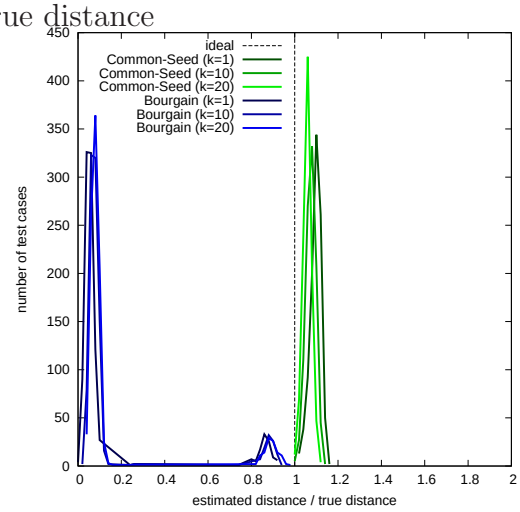


Figure 13: Distributions of the ratio of estimated distance to true distance for directed distance $d = 50$

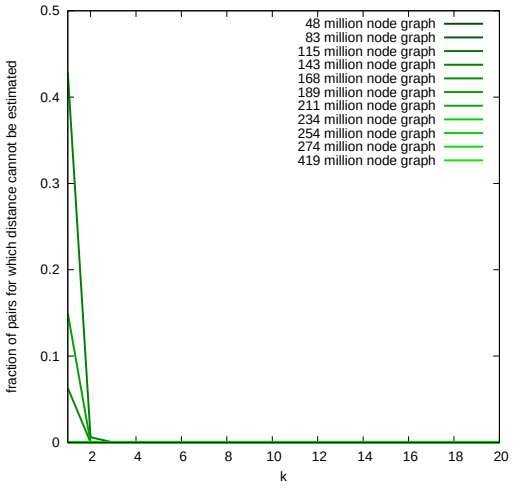


Figure 14: Fraction of non-covered (u, v) pairs for different k using Online-Common-Seed

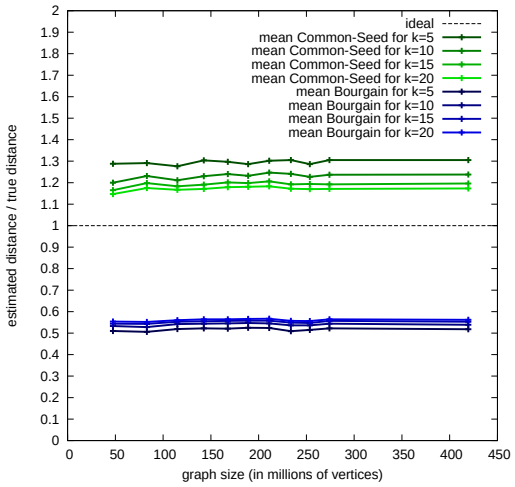


Figure 15: Effect of graph size on the estimated undirected distance

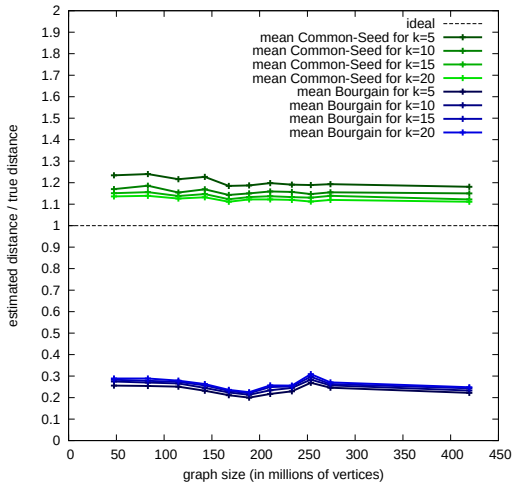


Figure 16: Effect of graph size in the estimated directed distance

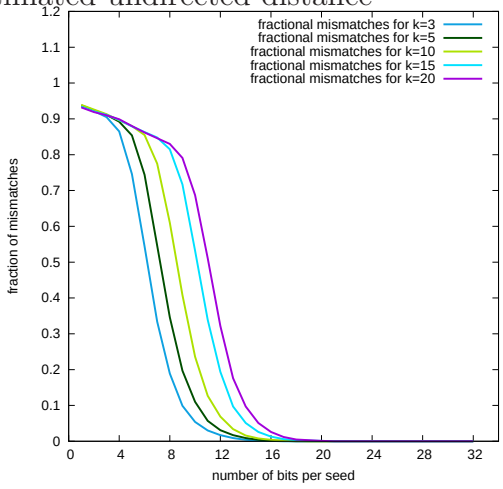


Figure 17: Effect of the number of bits per seed on the number of false positives - undirected case

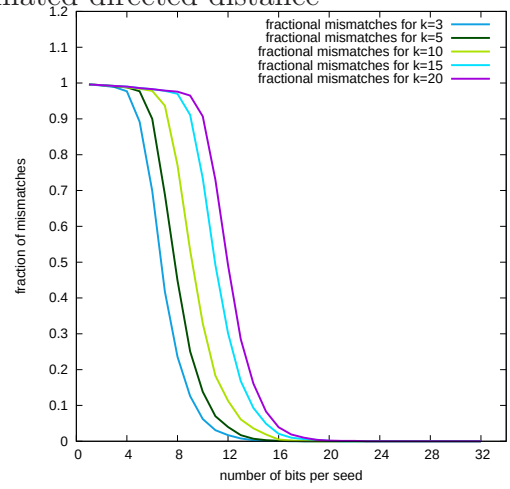


Figure 18: Effect of the number of bits per seed on the number of false positives - directed case

CHAPTER VII

BEST-ORDER STREAMING MODEL

In this chapter, we present a new model of computation called *stream checking* on graph problems where a space-limited verifier has to verify a proof sequentially (i.e., it reads the proof as a stream). Moreover, the proof itself is nothing but a reordering of the input data. This model has a close relationship to many models of computation in other areas such as data streams, communication complexity, and proof checking and could be used in applications such as cloud computing.

In this work [50] we focus on graph problems where the input is a sequence of edges. We show that checking if a graph has a perfect matching is impossible to do deterministically using small space. To contrast this, we show that randomized verifiers are powerful enough to check whether a graph has a perfect matching or is connected.

7.1 Related Work and Contributions

This work is motivated by three fundamental questions that arise in three widely studied areas in theoretical computer science - streaming algorithms, communication complexity, and proof checking. The first question is how efficient can space restricted streaming algorithms be. The second question, is whether the hardness of a communication problem holds for every partition of the input. Finally, in proof checking, the question is how many (extra) bits are needed for the verifier to establish a proof in a restricted manner.

Our model is motivated by the cloud computing domain where large computing tasks can be outsourced to massive parallel/cloud architectures. The cloud solves the problem and gets the answer back. How does one know the answer is correct? Is there

a way to verify the answer by asking the cloud to present a (streaming) proof? These questions are closely related to our model. Coincidentally, our model has connections with many previously studied models in many areas. We now continue with describing previous models studied specifically in the stream, computational complexity and proof checking domains and contrast them with our model.

Data Streams: The basic premise of streaming algorithms is that one is dealing with a humongous data set, too large to process in main memory. The algorithm has only sequential access to the input data; this called a *stream*. In certain settings, it is acceptable to allow the algorithm to perform multiple passes over the stream. However, for many applications, it is not feasible to perform more than a single pass. The general streaming algorithms framework has been studied extensively since the seminal work of Alon, Matias, Szegedy [7].

Models diverge in the assumptions made about what order the algorithm can access the input elements. The most stringent restriction on the algorithm is to assume that the input sequence is presented to the algorithm in an adversarial order. A slightly more relaxed setting, that has also been widely studied is where the input is assumed to be presented in randomized order [38, 79, 80]. However, even a simple problem like finding median, which was considered in the earliest paper in the area by Munro and Patterson [125], in both input orders, was shown recently [38] to require many passes even when the input is in a random order (to be precise, any $O(\text{polylog}\, n)$ algorithm requires $\Omega(\log \log n)$ passes). This might be undesirable.

More bad news: Graph problems are extremely hard when presented in an adversarial order. In [86], one of the earliest paper in this area, it was shown that many graph problems require prohibitively large amount of space to solve. It is confirmed by the more recent result [65] that most graph problems cannot be solved efficiently

in a few passes. Since then, new models have been proposed to overcome this obstruction. Feigenbaum et. al. [66] proposed a relaxation of the memory restriction in what is called the semi-stream model. Aggarwal et. al. [3] proposed that if the algorithm has a power to sort the stream in one pass then it is easier to solve some graph problems (although not in one or constant passes). Another model that has been considered is the W-Stream (write-stream) model [132, 53]. While the algorithm processes the input, it may also *write* a new stream to be read in the next pass.

We ask the following fundamental question:

> *If the input is presented in the best order possible, can we solve problems efficiently?*

A precise explanation is reserved for the models in Section 7.2; however, intuitively, this means that the algorithm processing the stream can decide on a *rule* on the order in which the stream is presented. We call this the best-order stream model. For an example, if the rule opted by the algorithm is to read the input in sorted order, then this is equivalent to the single pass sort stream model. Another example of a rule, for graphs presented as edge streams could be that the algorithm requires all edges incident on a vertex to be presented together. This is again equivalent to a graph stream model studied earlier called an incidence model (and corresponds to reading the rows of the adjacency matrix one after the other). A stronger rule could be that the algorithm asks for edges in some perfect matching followed by other edges. As we show in this work, this rule leads to checking if the graph has a perfect matching and as a consequence shows the difference between our model and the sort-stream model.

It would be nice to obtain a characterization of problems that can be solved by a poly-log space, single pass, best-order stream algorithm. Studying this model, like all other related streaming models, is likely to yield new insights and might lead to an improvement of worst case analysis and an adjustment of models.

Communication Complexity: Another closely related model is the communication complexity model [142, 102]. This model was extensively studied and found many applications in many areas. In the basic form of this model, two players, Alice and Bob, receive some input data and they want to compute some function together. The question is how much communication they have to make to accomplish the task. There are many variations of how the input is partitioned. The worst-case [104] and the best-case [129] partition models are two extreme cases that are widely studied over decades. The worst case asks for the partition that makes Alice and Bob communicate the most while the best case asks for the partition that makes the communication smallest. Moreover, even very recently, there is a study for another variation where the input is partitioned according to some known distribution (see, e.g., [37]). The main question is whether the hardness of a communication problem holds for almost every partition of the input, as opposed to holding for perhaps just a few atypical partitions.

The communication complexity version of our model (described in Section 7.2) asks the following similar question: Is the hardness of a communication problem holds for *every* partition of the input. Moreover, our model can be thought of as a more extreme version of the best-case partition communication complexity. We explain this in more details in Section 7.2.

Proof Checking: From a complexity theoretic standpoint, our model can be thought of as the case of proof checking where a polylog-space verifier is allowed to read the proof as a stream; additionally, the proof must be the input itself in a different order.

We briefly describe some work in the field of proof checking and its relation to our setting. The field of probabilistically checkable proofs (PCPs) [12, 14, 55] deals with verifier querying the proof at very few points (even if the data set is large and thus the proof) and using this to guarantee the proof with high probability. While several

variants of proof checking have been considered, we only state the most relevant ones. A result most related to our setting is by Lipton [110] where it showed that membership proofs for NP can be checked by probabilistic logspace verifiers that have one-way access to the proof and use $O(\log n)$ random bits. In other words, this result almost answers our question except that the proof is not the reordered input.

Another related result that compares streaming model with other models is by Feigenbaum et. al. [64] where the problem of testing and spot-checking on data streams is considered. They define sampling-tester and streaming-tester. A sampling-tester is allowed to sample some (but not all) of the input points, looking at them in any order. A streaming-tester, on the other hand is allowed to look at the entire input but only in a specific order. They show that some problems can be solved in a streaming-tester but not by a sampling-tester, while the reverse holds for other problems. Finally, we note that our model (when we focus on massive graphs) might remind some readers of the problem of property testing in massive graphs [74].

Notice that in all of the work above, there are two common themes. The first is verification using *small space*. The second is some form of *limited access* to the input. The limited access is either in the form of sampling from the input, limited communication, or some restricted streaming approach. Our model captures both these factors.

In this work, we partially answer whether there are efficient streaming algorithms when the input is in the best order possible. We give a negative answer to this question for the deterministic case and show an evidence of a positive answer for the randomized case. Our positive results are similar in spirit to W-stream and Sort-stream papers [3, 53, 132].

For the negative answer, we show that the space requirement is too large even for a simple answer of checking if a given graph has a perfect matching deterministically. In contrast, this problem, as well as the connectivity problem, can be solved efficiently

by randomized algorithms.

Organization The rest of the chapter is organized as follow. In Section 7.2 we describe our stream proof checking model formally and also define some of the other communication complexity models that are well-studied. The problem of checking for distinctness in a stream of elements is discussed in Section 7.3. This is a building block for most of our algorithms. The following section, Section 7.4 talks about how perfect matchings can be checked in our model. We discuss the problem of stream checking graph connectivity in Section 7.5. Our techniques can be extended to a wide class of graph problems such as checking for regular bipartiteness, non-bipartiteness, hamiltonian cycles etc. While we do not mention all details, we describe the key ideas for these problems in Section 7.6. Finally, we make concluding remarks in Section 7.7 by stating some insights drawn from this work, mention open problems and describe possible future directions.

7.2 Models

In this section we explain our main model and other related models that will be useful in subsequent sections.

7.2.1 Stream Proof Model

Recall the streaming model where an input is in some order $e_1, e_2, ..., e_m$ where m is the size of the input. Consider any function f that maps these input stream to $\{0, 1\}$. The goal of the typical one-pass streaming model is to calculate f using the smallest amount of memory possible.

In the *stream proof* model, we consider any function f that is order-independent. Our main question is how much space a one-pass streaming algorithm needs to compute f if the input is provided in the best order. Formally, for any function s of m and

any function f, we say that a language L determined by f is in the class STREAM-PROOF($s(m)$) if there exists an algorithm $\mathcal{A}$ using space at most $s(m)$ such that if $f(e_1, e_2, ..., e_m) = 1$ then there exists a permutation π such that $\mathcal{A}(e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(m)})$ answers 1; otherwise, $\mathcal{A}(e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(m)})$ answers 0 for *every* permutation π.

The other way to see this model is to consider the situation where there are two players in the setting, *prover* and *verifier*. The job of the prover is to provide the stream in some order so that the verifier can compute f using smallest amount of memory possible. We assume that the prover has unlimited power but restrict the verifier to read the input in a streaming manner.

The model above can be generalized to the following.

- STREAM(p, s): A class of problems that, when presented with best-order, can be checked by a deterministic streaming algorithm $\mathcal{A}$ using (p) passes $O(s)$ space.
- RSTREAM(p, s): A class of problems that, when presented with best-order, can be checked by a randomized streaming algorithm $\mathcal{A}$ using (p) passes $O(s)$ space and with correct probability more than 1/2.

It is important to point out that when the input is presented in a specified order, we still need to check that the adversary is not *cheating*. That is, we indeed need a way to verify that we receive the input based on the rule we asked for. This often turns out to be the difficult step.

To contrast this model with well-studied communication complexity models, we first define a new communication complexity model, magic-partition, that closely relates to our proof checking model.

7.2.2 Magic-Partition Communication Complexity

In this subsection, we define magic-partition communication complexity which will be the main tool to prove the lower bound of the best-order streaming model.

Recall that in the standard 2-player communication complexity, Alice and Bob gets input x and y and want to compute $f(x, y)$. We usually consider when the input is partitioned in an adversarial order, i.e., we partition input into x and y in such a way that Alice and Bob have to communicate as many bits as possible.

For the magic-partition communication complexity, we consider the case when x and y are partitioned in the best way possible. One way to think of this protocol is to imagine that there is an oracle who looks at the input and then decides how to divide the data between Alice and Bob so that they can compute f using smallest number of communicated bits. We restrict that the input data must be divided equally between Alice and Bob.

Let us consider an example. Suppose the input is a graph G. Alice and Bob might decide that the graph be broken down in topological sort order, and Alice receives the first half of the total edges, starting with edges incident on the vertices (traversing them in topological order). It is important to note the distinction that Alice and Bob actually have not seen the input; but they specify a *rule* by which to partition the input, when actually presented.

The following lemma is the key to prove our lower bound results.

Lemma 7.2.1. *For any function f, if the (deterministic) magic-partition communication complexity of f is at least s, for some s, then for any p and t such that $(2p-1)t < s$, $f \notin \textsc{Stream}(p, t)$.*

Proof. Suppose that the lemma is not true; i.e., f has magic-partition communication complexity at least s, for some s, but there is a best-order streaming algorithm $\mathcal{A}$ that computes f using p passes and t space such that $(2p-1)t < s$. Consider any input $e_1, e_2, ..., e_n$. Let π be a permutation such that $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(n)}$ is the best ordering of the input for $\mathcal{A}$. Then, define the partition of the magic-partition communication complexity by allocating $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$ to Alice and the rest to Bob.

Alice and Bob can simulate $\mathcal{A}$ as follows. First, Alice simulates $\mathcal{A}$ on $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$.

Then, she sends the data on memory to Bob. Then, Bob continues simulating $\mathcal{A}$ using data given by Alice (as if he simulates $\mathcal{A}$ on $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$ by himself). He then sends the data back to Alice and the simulation of the second pass of $\mathcal{A}$ begins.) Observe that this simulations need $2p-1$ rounds of communication and each round requires at most t bits. Therefore, Alice and Bob can compute f using $(2p-1)t < s$ bits, contradicting the original assumption. □ □

Note that this type of communication complexity must not be confused with the best-partition communication complexity (defined below). Also, the converse of the above lemma clearly does not hold.

We now describe some previously studied communication complexity models that resemble ours.

7.2.3 Related models

7.2.3.1 Best-case partition Communication Complexity

For this model, Alice and Bob can pick how to divide the data among them (must be half-half) *before* they see the input. Then, the adversary gives an input that make them communicate the most.

This model was introduced by Papadimitriou and Sipser [129] and heavily used for proving lower bounds for many applications (see [102] and references therein).

Similar to the best-partition communication complexity, this model makes many problems easier to solve than the traditional worst case communication complexity where the worst case input is assumed. One example is the set disjointness problem. In this problem, two n-bit vectors x and y that is a characteristic vector of two sets X and Y are given. Alice and Bob have to determine if $X \cap Y = \emptyset$.

In the worst case communication complexity, it is proved that Alice has to send roughly n bits to Bob when x is given to Alice and y is given to Bob. However, for the best-partition case, they can divide the input this way: $x_1, y_1, x_2, y_2, ..., x_{n/2}, y_{n/2}$

go to Alice and the rest go to Bob. This way, each of them can check the disjointness separately.

We note that this model is different from the magic-partition model in that, in this model the players have to pick how data will be divided before they see the input data. For example, if the data is the graph of n vertices then, for any edge (i, j), Alice and Bob have to decide who will get this edge if (i, j) is actually in the input data. However, in the magic-partition model, Alice and Bob can make a more complicated partitioning rule such as giving $(1, 2)$ to Alice *if* the graph is connected. (In other words, in the magic-partition model, Alice and Bob have an oracle that decide how to divide an input *after* he sees it).

One problem that separates these model is the connectivity problem. Hajnal et al. [83] showed that the best-case partition communication complexity of connectivity is $\Theta(n \log n)$. In contrast, we show that $O((\log n)^2)$ is possible in our model in this work.

7.2.3.2 Nondeterministic Communication Complexity

Alice and Bob receives x and y respectively. An oracle, who sees x and y, wants to convince them that "$f(x, y) = 1$". He does so by giving them a proof. Alice and Bob should be able to verify the proof with small amount of communication.

Example: $f(x, y) = 1$ if $x \neq y$ where x and y are n-bit strings. The proof is simply the number i where $x_i \neq y_i$. Alice and Bob can check the proof by exchanging x_i and y_i. If $x = y$ then there is no proof and Alice and Bob can always detect the fake proof.

This model is different from our model because our model has no proof but the oracle's job is to help Alice and Bob find the answer (whether $f(x, y)$ is 0 or 1) by appropriately partitioning the input.

7.3 Checking distinctness

In this section, we consider the following problem which is denoted by DISTINCT. Given a stream of n numbers $a_1, a_2, ..., a_n$ where $a_i \in \{1, 2, ..., n\}$. We want to check if every number appears exactly once (i.e., no duplicate). This problem appears as a main component in all the problems we considered and we believe that it will appear in every problem.

Our goal in this section is to find a one-pass algorithm for this problem. An algorithm for this problem will be an important ingredient of all algorithm we consider in this work. In this section, we show that 1) any deterministic algorithm for this problem needs $\Omega(n)$ space, and 2) there is a randomized algorithm that solves this problem in $O(\log n)$ space with error probability $\frac{1}{n}$.

7.3.1 Space lower bound of deterministic algorithms

Since checking for distinctness is equivalent to checking if there is a duplicate, a natural problem to use as a lower bound is the *set disjointness problem*. We define a variation of this problem called *full set disjointness problem*, denoted by F-DISJ.

For this problem, a set $X \subseteq N$ is given to Alice and $Y \subseteq N$ is given to Bob where $N = \{1, 2, 3, ..., n\}$ and $|X| + |Y| = n$.[1]

Now we show that F-DISJ is hard for the deterministic case. The proof is the same as the proof of the set disjointness problem.

Theorem 7.3.1. *The communication complexity of* F-DISJ *is* $\Omega(n)$.

Proof. Consider the fooling set $F = \{(A, \bar{A}) \ : \ \forall A \subseteq N\}$. Since $|F| = 2^n$, the number of bits needed to sent between Alice and Bob is at least $\log |F| = \Omega(n)$. □ □

[1]Note that this problem is different from the well-known set disjointness problem in that we require $|X| + |Y| = n$. Although the two problems are very similar, they are different in that the set disjointness problem has $\Omega(n)$ randomized algorithm while the F-DISJ has an $O(\log n)$ randomized protocol (shown in the next section). We also note that the lower bound of another related problem called k-disjointness problem ([102, example 2.12] and [84]) does not imply our result neither.

The communication complexity lower bound of F-DISJ implies the space lower bound of DISTINCT.

Corollary 7.3.2. *Any deterministic algorithm for* DISTINCT *needs* $\Omega(n)$ *space.*

This lower bound is for worst-case input. The reason we mention this here is because this is an inherent difficulty in our algorithms. Our randomized algorithms use randomness only to get around this step of checking distincness.

7.3.2 Randomized algorithm

In this subsection we present a randomized one-pass algorithm that solves this problem using $O(\log n)$ space. This algorithm is based on the *Fingerprinting Sets* technique introduced by Lipton [109, 110]. Roughly speaking, given a multi-set $\{x_1, x_2, ..., x_k\}$, its *fingerprint* is defined to be

$$\Pi_{i=1}^{k}(x_i + r) \mod p$$

where p is a random prime and $r \in \{0, 1, ..., p-1\}$. We use the following property of the fingerprints.

Theorem 7.3.3. *[110] Let* $\{x_1, x_2, ..., x_k\}$ *and* $\{y_1, y_2, ..., y_k\}$ *be two multi-sets. If the two sets are equal then their fingerprints are always the same. Moreover, if they are unequal, the probability that they get the same fingerprints is at most*

$$O(\frac{\log b + \log m}{bm} + \frac{1}{b^2 m})$$

where all numbers are b-bit numbers and $m = \max(k, l)$ *provided that the prime p is selected randomly from interval*

$$[(bm)^2, 2(bm)^2].$$

Now, to check if $a_1, a_2, ..., a_n$ are all distinct, we simply check if the fingerprints of $\{a_1, a_2, ..., a_n\}$ and $\{1, 2, ..., n\}$ are the same. Here, $b = \log n$ and $m = n$. Therefore, the error probability is at most $1/n$.

Remark: We note that the fingerprinting sets can be used in our motivating application above. That is, when the cloud compute sends back a graph as a proof, we have to check whether this "proof" graph is the same as the input graph we sent. This can be done using the fingerprinting set. This enables us to concentrate on checking the stream without worrying about this issue in the rest of the work.

We also note that the recent result by Gopalan et al. [76] can be modified to solve DISTINCT as well.

7.4 Perfect Matching

This section is devoted to the study of perfect matchings. We discuss lower bounds as well as upper bounds.

Problem: Given the edges of a graph G in a streaming manner $e_1, e_2, ..., e_m$, we want to compute $f(e_1, ..., e_m)$ which is 1 if and only if G has a perfect matching. Let n be the number of vertices. We assume that the vertices are labeled $1, 2, ..., n$.

We now present the main upper bound of this section and follow it up with the checking protocol in the proof.

Theorem 7.4.1. *Problem of determining if there exists a perfect matching can be done by a randomized algorithm in $O(\log n)$ space best-order stream checking.*

Proof. **Protocol:** The prover sends $n/2$ edges of a perfect matching to the verifier, followed by the "sign" which can be implemented by flipping the order of vertices in the last edge. Then the prover sends the rest edges. The verifier has to check three things.

1. Find out n by counting the number of edges before the "sign" is given.

2. Check if the first $n/2$ edges form a perfect matching. This can be done by checking if the sum of the labels of vertices in the first $n/2$ edges equals $1+2+3+...+n$.

3. Check if there are n vertices. This is done by checking that the maximum vertex label is at most n.

The verifier outputs 1 if the input passes all the above tests. The correctness of this protocol is straightforward. □ □

In the next subsection, we present a lower bound.

7.4.1 Hardness

We show that deterministic algorithms have $\Omega(n)$ lower bound if the input is reordered in an explicit way; i.e., each edge cannot be split. This means that an edge is either represented in the form (a, b) or (b, a). The proof follows by a reduction from the best-partition-after-input communication complexity (see Section 7.2) of the same problem by using Lemma 7.2.1.

Theorem 7.4.2. *If the input can be reordered only in an explicit way then any deterministic algorithm solving the perfect matching problem needs $\Omega(n)$ space.*

Proof. Let n be an even integer. Let $f(n)$ denote the number of matchings in the complete graph K_n. Observe that $f(n) = \frac{n!}{(n/2)!2^{n/2}}$. Denote these matchings by $M_1, M_2, ..., M_{f(n)}$. Let $\mathcal{P}$ be any best-order-after-partition protocol. For any integer i, let A_i and B_i be the best partition of M_i according to $\mathcal{P}$ (A_i and B_i are sent to Alice and Bob respectively). Observe that for any i, there are at most $f(n/2)^2$ matchings that vertices are partitioned the same way as M_i. (I.e., if we define $C_i = \{v \in V \mid \exists e \in A_i \ s.t. \ v \in e\}$ then for any i, $|\{j \ |C_i = C_j\}| \leq f(n/2)^2$.) This is because $n/2$ vertices on each side of the partition can make $f(n/2)$ different matchings.

Therefore, the number of matchings such that the vertices are divided differently is at least

$$\frac{n!}{(n/2)!2^{n/2}}(\frac{(n/4)!2^{n/4}}{(n/2)!})^2 = \binom{n}{n/2}/\binom{n/2}{n/4} \geq \binom{n/2}{n/4}.$$

(The last inequality follows from the fact that $\binom{n}{n/2}$ is the number of $n/2$-subsets of $\{1, 2, ..., n\}$ and $\binom{n/2}{n/4}^2$ is the number of parts of these subsets.)

In particular, if we let $M_{i_1}, ..., M_{i_t}$, where $t = \binom{n/2}{n/4}$, be such matchings then for any $j \neq k$, (A_{i_j}, B_{i_k}) is not a perfect matching.

Now, let $t' = \log t$. Note that $t' = \Omega(n)$. Consider the problem $\text{EQ}_{t'}$ where Alice and Bob each gets a t'-bit vector x and y, respectively. They have to output 1 if $x = y$ and 0 otherwise. By [102, example 1.21], $D(\text{EQ}_{t'}) \geq t' + 1 = \Omega(n)$.

Now we reduce $\text{EQ}_{t'}$ to our problem: Map x to M_{i_x} and y to M_{i_y}. Now, $x = y$ if and only if (M_{i_x}, M_{i_y}) is a perfect matching. This shows that the best-partition-after-read communication complexity of the matching problem is $\Omega(n)$. □ □

Note the the above lower bound is asymptotically tight since there is an obvious protocol where Alice sends Bob all vertices she has(using $O(n)$ bits of communication).

7.5 Graph Connectivity

Graph connectivity is perhaps the most basic property that one would like to check. However, even graph connectivity does not admit space-efficient algorithms in traditional streaming models. There is an $\Omega(n)$ lower bound for randomized algorithms. To contrast this, we show that allowing the algorithm the additional power of requesting the input in a specific order allows for very efficient, $O((\log n)^2)$ space algorithms for testing connectivity.

Problem: We consider a function where the input is a set of edges and $f(e_1, e_2, ..., e_m) =$ 1 if and only if G is connected. As usual, let n be the number of vertices of G. As

before, we assume that vertices are labeled $1, 2, 3, ..., n$.

We We will prove the following theorem.

Theorem 7.5.1. *Graph connectivity can be probabilistically checked using $O((\log n)^2)$ space in the pure model.*

Proof. We use the following lemma which is easy to prove.

Lemma 7.5.2. *For any graph G of n edges, G is connected if and only if there exists a vertex v and trees $T_1, T_2, ..., T_q$ such that for all i,*

- *there exists a unique vertex $u_i \in V(T_i)$ such that $uv \in E(T_i)$, and*
- *$|V(T_i)| \leq 2n/3$ for all i.*

Suppose G is connected, i.e., G is a tree. Let v and $T_1, T_2, ..., T_q$ be as in the lemma. Define the order of G to be

$$Order(G) = vu_1, Order(T_1'), vu_2, Order(T_2'), ..., vu_q, Order(T_q')$$

where $T_i' = T_i \setminus \{vu_i\}$. Note that T_i' is a connected tree and so we present edges of T_i' recursively.

Now, when edges are presented in this order, the checker can check if the graph is connected as follows. First, the checker reads vu_1. He checks if T_1' is connected by running the algorithm recursively. Note that he stops checking T_1' once he sees vu_2. Next, he repeats with vu_2 and T_2' and so on.

The space needed is for vu_i and for checking T_i'. I.e., $space(|G|) = space(\max_i |T_i|) + O(\log n)$. That is, $space(n) \leq space(2n/3) + O(\log n)$. This gives the claimed space bound.

Note that the checker has to make sure every vertex appears in the graph. He does so by applying result in Section 7.3 once to each vertex v used as a root (as in above) and all leaf nodes of the tree.

Also note that if G is not connected then such ordering cannot be made and the algorithm above will detect this fact. □ □

7.6 Further Results

The previous sections give us a flavor of the results that can be obtained in this model. We describe a few more and mention the intuition behind the corresponding protocols.

7.6.1 Bipartite k-Regular graph

The point is that a k-regular bipartite graph can be decomposed into k disjoint sets of perfect matchings. So the adversary can do this and present each of the perfect matchings one after the other. Now our previously described algorithm can be used to verify each perfect matching. In addition, a fairly simple algorithm can take care of verifying that we indeed receive k different sets (and to also know when one perfect matching ends and the new one is presented).

7.6.2 Hamiltonian cycle

It can be shown that Hamiltonian-Cycle$\in$ RStream$(1, \log n)$. The intuition is for the protocol to request the hamiltonian cycle first (everything else is ignored). The checker then checks if the first n edges presented indeed form a cycle; this requires two main facts. First that every two consecutive edges share a vertex, and the n-th edge shares a specific vertex with the first. This is easy. The second key step is to check that these edges indeed span all n vertices (and not go through same vertex more than once). This can be done by using the set distinctness approach.

7.6.3 Non-bipartiteness

Non-bipratitiness of graphs can again be checked in our model by requesting the adversary to present an odd length cycle. Verifying that this is indeed a cycle and

that it is of odd length is again done in a manner very similar to verifying hamiltonian cycle.

We do not have an algorithm to verify general *bipartiteness* of graphs and leave it as an open question.

7.7 Concluding Remarks

This work described a new model of stream checking, that lies at the intersection of extremely well-studied and foundational fields of computer science. Specifically, the model connects several settings relating to proof checking, communication complexity, and streaming algorithms. The motivation for this work, however, draws from recent growth in data sizes and the advent of powerful cloud computing architectures and services. The question we ask is, can verification of certain properties (on any input) be accompanied with a streaming proof of the fact? The checker should be able to verify that the prover is not cheating. We show that if the checker (or algorithm in the streaming algorithms setting) is given the power of choosing a specific rule for the prover to send the input, then this is in fact possible much more efficiently than in previous models.

While non-obvious, our algorithms and proofs are fairly simple. However, the nice aspect is that it uses several interesting techniques and areas such as fingerprinting, and covert channels. Fingerprinting is used in a crucial way to randomly test for distinctness of a set of elements presented as a stream. The protocol between the prover and check also allows for covert communication (which gives covert channels a positive spin as opposed to previous studies in security and cryptography). While the prover is only allowed to send the input, re-ordered, the prover is able to encode extra bits of information with the special ordering requested by the checker. The difficulty in most of our proof techniques is in how the checker or algorithm verifies that the prover or adversary is sending the input order as requested.

We have given $O(\text{polylog}\, n)$ space algorithms for problems that previously, in the streaming model, had no $O(n)$ algorithms. There are still a lot of problems in graph theory that remain to be investigated. A nice direction is to consider testing for graph minors, which could in turn yield efficient methods for testing planarity and other properties that exclude specific minors. We have some work in progress in this direction. Apart from the study of our specific model, we believe that the results and ideas presented in this work could lead to improved algorithms in previously studied settings as well as yield new insights to the complexity of the problems.

REFERENCES

[1] "Amazon elastic compute cloud (amazon ec2)."

[2] Adamic, L. A., Lukose, R. M., Puniyani, A. R., and Huberman, B. A., "Search in power-law networks," *Physical Review*, vol. 64, 2001.

[3] Aggarwal, G., Datar, M., Rajagopalan, S., and Ruhl, M., "On the streaming model augmented with a sorting primitive," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 540–549, 2004.

[4] Ahn, K. J. and Guha, S., "Sparsification Algorithm for Cut Problems Semi-streaming Model," in *http://arxiv.org/abs/0902.0140*, 2009.

[5] Aldous, D., "A random walk construction of uniform random spanning trees and uniform labelled trees," *SIAM Journal on Discrete Mathematics*, vol. 3, no. 4, pp. 450–465, 1990.

[6] Aleliunas, R., Karp, R. M., Lipton, R. J., Lovász, L., and Rackoff, C., "Random walks, universal traversal sequences, and the complexity of maze problems," in *FOCS*, pp. 218–223, 1979.

[7] Alon, N., Matias, Y., and Szegedy, M., "The space complexity of approximating the frequency moments," *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, 1999.

[8] Alon, N., "Eigenvalues and expanders," in *Combinatorica*, pp. 6(2) 83–96, 1986.

[9] Alon, N., Avin, C., Koucký, M., Kozma, G., Lotker, Z., and Tuttle, M. R., "Many random walks are faster than one," in *SPAA*, pp. 119–128, 2008.

[10] Andersen, R., Chung, F. R. K., and Lang, K. J., "Local graph partitioning using pagerank vectors," in *Proc. of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 475–486, 2006.

[11] Armoni, R., Ta-Shma, A., Wigderson, A., and Zhou, S., "$sl \leq l^{4/3}$," in *ACM Symposium on Theory of Computing*, pp. 230–239, 1997.

[12] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M., "Proof verification and the hardness of approximation problems," *J. ACM*, vol. 45, no. 3, pp. 501–555, 1998.

[13] Arora, S., Rao, S., and Vazirani, U. V., "Expander flows, geometric embeddings and graph partitioning," in *STOC*, pp. 222–231, 2004.

[14] Arora, S. and Safra, S., "Probabilistic checking of proofs: A new characterization of np," *J. ACM*, vol. 45, no. 1, pp. 70–122, 1998. Also appeared in FOCS'92.

[15] Awerbuch, B., Baratz, A., and Peleg, D., "Cost-sensitive analysis of communication protocols," in *9th ACM PODC*, pp. 177–187, 1990.

[16] Baala, H., Flauzac, O., Gaber, J., Bui, M., and El-Ghazawi, T., "A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks," *Journal of Parallel and Distributed Computing*, vol. 63, no. 1, pp. 97–104, 2003.

[17] Bar-Ilan, J. and Zernik, D., "Random leaders and random spanning trees," in *Proceedings of the 3rd International Workshop on Distributed Algorithms(later called DISC)*, (London, UK), pp. 1–12, Springer-Verlag, 1989.

[18] Bar-Yossef, Z., Kumar, R., and Sivakumar, D., "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *In Proc. ACM-SIAM Symposium on Discrete Algorithms*, pp. 623–632, 2002.

[19] Bartal, Y., "On approximating arbitrary metrices by tree metrics," in *STOC*, pp. 161–168, 1998.

[20] Baswana, S., "Streaming algorithm for graph spanners - single pass and constant processing time per edge," *Inf. Process. Lett.*, vol. 106, no. 3, pp. 110–114, 2008.

[21] Batu, T., Fischer, E., Fortnow, L., Kumar, R., Rubenfeld, R., and White, P., "Testing random variables for independence and identity," in *Proc. of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 442–451, 2001.

[22] Benczúr, A. A. and Karger, D. R., "Approximating s-t Minimum Cuts in $\tilde{O}(n^2)$ Time," in *STOC*, 1996.

[23] Bernard, T., Bui, A., and Flauzac, O., "Random distributed self-stabilizing structures maintenance," in *ISSADS*, pp. 231–240, 2004.

[24] Bharambe, A. R., Agrawal, M., and Seshan, S., "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM*, pp. 353–366, 2004.

[25] Bhatt, S. N. and Leighton, F. T., "A Framework for Solving VLSI Graph Layout Problems," *JCSS*, vol. 28, no. 2, 1984.

[26] Bhuvanagiri, L. and Ganguly, S., "Estimating entropy over data streams," in *European Symposium on Algorithms (ESA)*, pp. 148–159, 2006.

[27] Bhuvanagiri, L., Ganguly, S., Kesh, D., and Saha, C., "Simpler algorithm for estimating frequency moments of data streams," in *Proc of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 708–713, 2006.

[28] Boppana, R., "Eigenvalues and graph bisection: an average case analysis," in *FOCS*, 1987.

[29] Borgs, C., Chayes, J. T., Mahdian, M., and Saberi, A., "Exploring the community structure of newsgroups," in *KDD*, pp. 783–787, 2004.

[30] Bourgain, J., "On Lipschitz embeddings of finite metric spaces in hilbert space," *Israel Journal of Mathematics*, vol. 52(1-2), pp. 46–52, 1985.

[31] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th international conference on World Wide Web*, pp. 107–117, 1998.

[32] Broder, A. Z., "Generating random spanning trees," in *FOCS*, pp. 442–447, 1989.

[33] Broder, A. Z., "Identifying and filtering near-duplicate documents," in *CPM*, pp. 1–10, 2000.

[34] Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G., "Syntactic clustering of the web," *Computer Networks*, vol. 29, no. 8, pp. 1157–1166, 1997.

[35] Bui, M., Bernard, T., Sohier, D., and Bui, A., "Random walks in distributed computing: A survey," in *IICS*, pp. 1–14, 2004.

[36] Buriol, L. S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohle, C., "Counting triangles in data streams," in *PODS*, pp. 253–262, 2006.

[37] Chakrabarti, A., Cormode, G., and McGregor, A., "Robust lower bounds for communication and stream computation," in *STOC*, pp. 641–650, 2008.

[38] Chakrabarti, A., Jayram, T. S., and Patrascu, M., "Tight lower bounds for selection in randomly ordered streams," in *SODA*, pp. 720–729, 2008.

[39] Chung, F., *Spectral Graph Theory*. Providence, RI, USA: American Mathematical Society, 1997.

[40] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U., "Reachability and distance queries via 2-hop labels," in *SODA*, pp. 937–946, 2002.

[41] Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., and Peleg, D., "Labeling schemes for tree representation," *Algorithmica*, vol. 53, no. 1, pp. 1–15, 2009.

[42] Cooper, B. F., "Quickly routing searches without having to move content," in *IPTPS*, pp. 163–172, 2005.

[43] Cooper, C., Frieze, A., and Radzik, T., "Multiple random walks in random regular graphs," in *Preprint*, 2009.

[44] Coppersmith, D., Tetali, P., and Winkler, P., "Collisions among random walks on a graph," *SIAM J. Discret. Math.*, vol. 6, no. 3, pp. 363–374, 1993.

[45] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., "Introduction to algorithms," *MIT Press and McGraw-Hill*, vol. 24.3, pp. 595–601, 2001.

[46] Cormode, G. and Muthukrishnan, S., "Space efficient mining of multigraph streams," in *In ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pp. 271–282, 2005.

[47] Das Sarma, A., Gollapudi, S., Najork, M., and Panigrahy, R., "A sketch-based distance oracle for web-scale graphs," in *WSDM*, pp. 401–410, 2010.

[48] Das Sarma, A., Gollapudi, S., and Panigrahy, R., "Estimating pagerank on graph streams," in *PODS*, pp. 69–78, 2008.

[49] Das Sarma, A., Gollapudi, S., and Panigrahy, R., "Sparse cut projections in graph streams," in *ESA*, pp. 480–491, 2009.

[50] Das Sarma, A., Lipton, R. J., and Nanongkai, D., "Best-order streaming model," in *TAMC*, pp. 178–191, 2009.

[51] Das Sarma, A., Nanongkai, D., and Pandurangan, G., "Fast distributed random walks," in *PODC*, pp. 161–170, 2009.

[52] Das Sarma, A., Nanongkai, D., Pandurangan, G., and Tetali, P., "Efficient distributed random walks with applications," in *PODC*, 2010.

[53] Demetrescu, C., Finocchi, I., and Ribichini, A., "Trading off space for passes in graph streaming problems," in *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, (New York, NY, USA), pp. 714–723, ACM, 2006.

[54] Demetrescu, C., Goldberg, A. V., and Johnson, D. S., "Implementation challenge for shortest paths," in *Encyclopedia of Algorithms*, 2008.

[55] Dinur, I., "The pcp theorem by gap amplification," *J. ACM*, vol. 54, no. 3, p. 12, 2007. Also appeared in STOC'06.

[56] Dolev, S. and Tzachar, N., "Spanders: Distributed spanning expanders," *Dept. of Computer Science, Ben-Gurion University, TR-08-02*, 2007.

[57] Dolev, S., Schiller, E., and Welch, J. L., "Random walk for self-stabilizing group communication in ad hoc networks," *IEEE Trans. Mob. Comput.*, vol. 5, no. 7, pp. 893–905, 2006. also in PODC'02.

[58] Dong, W., Charikar, M., and Li, K., "Asymmetric distance estimation with sketches for similarity search in high-dimensional spaces," in *SIGIR*, pp. 123–130, 2008.

[59] Dubhashi, D., Grandioni, F., and Panconesi, A., "Distributed algorithms via lp duality and randomization," in *Handbook of Approximation Algorithms and Metaheuristics*, 2007.

[60] Elkin, M., "An overview of distributed approximation," *ACM SIGACT News Distributed Computing Column*, vol. 35, pp. 40–57, December 2004.

[61] Elkin, M., "Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem," in *Proceedings of Symposium on Theory of Computing (STOC)*, June 2004.

[62] Fakcharoenphol, J., Rao, S., and Talwar, K., "A tight bound on approximating arbitrary metrics by tree metrics," *J. Comp. Syst. Sci.*, vol. 69, no. 3, pp. 485–497, 2004.

[63] Feige, U., "A spectrum of time-space trade-offs for undirected s-t connectivity," *Journal of Computer and System Sciences*, vol. 54, no. 2, pp. 305–316, 1997.

[64] Feigenbaum, J., Kannan, S., Strauss, M., and Viswanathan, M., "Testing and spot-checking of data streams (extended abstract)," in *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 165–174, Society for Industrial and Applied Mathematics, 2000.

[65] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J., "Graph distances in the streaming model: the value of space," in *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 745–754, Society for Industrial and Applied Mathematics, 2005.

[66] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J., "On graph problems in a semi-streaming model," *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 207–216, 2005.

[67] Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., and Svitkina, Z., "On the complexity of processing massive, unordered, distributed data," in *CoRR abs/cs/0611108*, 2006.

[68] Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L., "Peer-to-peer membership management for gossip-based protocols," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 139–149, 2003.

[69] Garay, J., Kutten, S., and Peleg, D., "A sublinear time distributed algorithm for minimum-weight spanning trees," *SIAM J. Comput.*, vol. 27, pp. 302–316, 1998.

[70] Gavoille, C., Peleg, D., Pérennes, S., and Raz, R., "Distance labeling in graphs," *J. Algorithms*, vol. 53, no. 1, pp. 85–112, 2004.

[71] Gkantsidis, C., Goel, G., Mihail, M., and Saberi, A., "Towards topology aware networks," in *IEEE INFOCOM*, 2007.

[72] Gkantsidis, C., Mihail, M., and Saberi, A., "Hybrid search schemes for unstructured peer-to-peer networks," in *INFOCOM*, pp. 1526–1537, 2005.

[73] Goldberg, A. V., "Point-to-point shortest path algorithms with preprocessing," in *SOFSEM (1)*, pp. 88–102, 2007.

[74] Goldreich, O., "Property testing in massive graphs," pp. 123–147, 2002.

[75] Goldreich, O. and Ron, D., "Property Testing in Bounded Degree Graphs," in *STOC*, pp. 406–415, 1997.

[76] Gopalan, P. and Radhakrishnan, J., "Finding duplicates in a data stream," in *SODA*, pp. 402–411, 2009.

[77] Goyal, N., Rademacher, L., and Vempala, S., "Expanders via random spanning trees," in *SODA*, pp. 576–585, 2009.

[78] Greenwald, M. and Khanna, S., "Space-efficient online computation of quantile summaries," in *In ACM International Conference on Management of Data, SIGMOD*, pp. 58–66, 2001.

[79] Guha, S. and McGregor, A., "Approximate quantiles and the order of the stream," in *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 273–279, ACM, 2006.

[80] Guha, S. and Mcgregor, A., "Lower bounds for quantile estimation in random-order and multi-pass streaming," in *International Colloquium on Automata, Languages and Programming (ICALP*, pp. 704–715, 2007.

[81] Guha, S. and McGregor, A., "Space-efficient sampling," in *In AISTATS*, pp. 169–176, 2007.

[82] Guha, S., McGregor, A., and Venkatasubramanian, S., "Streaming and sublinear approximation of entropy and information distances," in *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pp. 733–742, 2006.

[83] Hajnal, A., Maass, W., and Turán, G., "On the communication complexity of graph properties," in *STOC*, pp. 186–191, 1988.

[84] Håstad, J. and Wigderson, A., "The randomized communication complexity of set disjointness," *Theory of Computing*, vol. 3, no. 1, pp. 211–219, 2007.

[85] Hastings, W. K., "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, pp. 97–109, April 1970.

[86] Henzinger, M. R., Raghavan, P., and Rajagopalan, S., "Computing on data streams," vol. 50, pp. 107–118, 1999.

[87] Indyk, P. and Woodruff, D. P., "Optimal approximations of the frequency moments of data streams," in *IEEE Symposium on Foundations of Computer Science, FOCS*, pp. 283–292, 2003.

[88] Indyk, P., "Algorithms for dynamic geometric problems over data streams," in *ACM Symposium on Theory of Computing, STOC*, pp. 373–380, 2004.

[89] Israeli, A. and Jalfon, M., "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *PODC*, pp. 119–131, 1990.

[90] Jerrum, M. and Sinclair, A., "Approximating the permanent," *SIAM Journal of Computing*, vol. 18, no. 6, pp. 1149–1178, 1989.

[91] Jowhari, H. and Ghodsi, M., "New streaming algorithms for counting triangles in graphs," in *COCOON*, pp. 710–716, 2005.

[92] Karger, D. R., "Minimum cuts in near-linear time," *J. ACM*, vol. 47, no. 1, pp. 46–76, 2000.

[93] Karger, D. R. and Ruhl, M., "Simple efficient load balancing algorithms for peer-to-peer systems," in *SPAA*, pp. 36–43, 2004.

[94] Katz, M., Katz, N. A., Korman, A., and Peleg, D., "Labeling schemes for flow and connectivity," *SIAM J. Comput.*, vol. 34, no. 1, pp. 23–40, 2004.

[95] Kelner, J. and Madry, A., "Faster generation of random spanning trees," in *IEEE FOCS*, 2009.

[96] Kempe, D. and McSherry, F., "A decentralized algorithm for spectral analysis," *Journal of Computer and System Sciences*, vol. 74(1), pp. 70–83, 2008.

[97] Kempe, D., Kleinberg, J. M., and Demers, A. J., "Spatial gossip and resource location protocols," in *STOC*, pp. 163–172, 2001.

[98] Khan, M., Kuhn, F., Malkhi, D., Pandurangan, G., and Talwar, K., "Efficient distributed approximation algorithms via probabilistic tree embeddings," in *Proc. 27th ACM Symp. on Principles of Distributed Computing (PODC)*, 2008.

[99] Khan, M. and Pandurangan, G., "A fast distributed approximation algorithm for minimum spanning trees," *Distributed Computing*, vol. 20, pp. 391–402, 2008.

[100] Kleinberg, J. M., "Authoritative sources in a hyperlinked environment," in *SODA*, pp. 668–677, 1998.

[101] Kleinberg, J. M., "The small-world phenomenon: an algorithm perspective," in *STOC*, pp. 163–170, 2000.

[102] Kushilevitz, E. and Nisan, N., *Communication complexity*. New York, NY, USA: Cambridge University Press, 1997.

[103] Kutten, S. and Peleg, D., "Fast distributed construction of k-dominating sets and applications," *J. Algorithms*, vol. 28, pp. 40–66, 1998.

[104] Lam, T. W. and Ruzzo, W. L., "Results on communication complexity classes," *J. Comput. Syst. Sci.*, vol. 44, no. 2, pp. 324–342, 1992. Also appeared in Structure in Complexity Theory Conference 1989.

[105] Law, C. and Siu, K.-Y., "Distributed construction of random expander networks," in *INFOCOM*, 2003.

[106] Lempel, . and Moran, S., "Salsa: the stochastic approach for link-structure analysis," *ACM Trans. Inf. Syst.*, 2001.

[107] Leskovec, J., Dumais, S., and Horvitz, E., "Web projections: learning from contextual subgraphs of the web," in *WWW*, (New York, NY, USA), pp. 471–480, ACM, 2007.

[108] Leskovec, J. and Faloutsos, C., "Sampling from large graphs," in *KDD*, pp. 631–636, 2006.

[109] Lipton, R. J., "Fingerprinting sets," cs-tr-212-89, Princton University, 1989.

[110] Lipton, R. J., "Efficient checking of computations," in *STACS*, pp. 207–215, 1990.

[111] Loguinov, D., Kumar, A., Rai, V., and Ganesh, S., "Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience," in *SIGCOMM*, pp. 395–406, 2003.

[112] Lovász, L. and Simonovits, M., "Random Walks in a Convex Body and an Improved Volume Algorithm," *Structures*, 1993.

[113] Lovász, L. and Simonovits, M., "The Mixing Rate of Markov Chains, an Isoperimetric Inequality, and Computing the Volume," in *FOCS*, pp. 346–354, 1990.

[114] Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S., "Search and replication in unstructured peer-to-peer networks," in *ICS*, pp. 84–95, 2002.

[115] Lynch, N., *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[116] Lyons, R., "Asymptotic enumeration of spanning trees," *Combinatorics, Probability & Computing*, vol. 14, no. 4, pp. 491–522, 2005.

[117] Manku, G., Rajagopalan, S., and Lindsay, B., "Randomized sampling techniques for space efficient online computation of order statistics of large datasets," in *In ACM SIGMOD International Conference on Management of Data*, pp. 251–262, 1999.

[118] Manokaran, R., Naor, J., Raghavendra, P., and Schwartz, R., "SDP gaps and UGC hardness for multiway cut, 0-extension, and metric labeling," in *STOC*, pp. 11–20, 2008.

[119] Matousek, J., "On the distortion required for embedding finite metric spaces into normed spaces," *Israel Journal of Mathematics*, vol. 93(1), pp. 333–344, 1996.

[120] McGregor, A., "Finding graph matchings in data streams," in *In APPROX-RANDOM*, pp. 170–181, 2005.

[121] McSherry, F., "A uniform approach to accelerated pagerank computation," in *WWW*, pp. 575–582, 2005.

[122] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E., "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[123] Mitzenmacher, M. and Upfal, E., *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. New York, NY, USA: Cambridge University Press, 2005.

[124] Morales, R. and Gupta, I., "Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems," in *ICDCS*, p. 55, 2007.

[125] Munro, J. I. and Paterson, M., "Selection and sorting with limited storage," *Theor. Comput. Sci.*, vol. 12, pp. 315–323, 1980. Also appeared in FOCS'78.

[126] Muthukrishnan, S. and Pandurangan, G., "The bin-covering technique for thresholding random geometric graph properties," in *ACM SODA*, 2005. Journal version to appear in Journal of Computer and System Sciences.

[127] Najork, M., "The scalable hyperlink store," in *Hypertext*, pp. 89–98, 2009.

[128] Pandurangan, G. and Khan, M., "Theory of communication networks," in *Algorithms and Theory of Computation Handbook, Second Edition*, CRC Press, 2009.

[129] Papadimitriou, C. H. and Sipser, M., "Communication complexity," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 260–269, 1984. Also appeared in STOC'82.

[130] Peleg, D. and Rabinovich, V., "A near-tight lower bound on the time complexity of distributed mst construction," in *Proc. of the 40th IEEE Symp. on Foundations of Computer Science*, pp. 253–261, 1999.

[131] Peleg, D., *Distributed computing: a locality-sensitive approach*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[132] Ruhl, J. M., *Efficient algorithms for new computational models*. PhD thesis, 2003. Supervisor-David R. Karger.

[133] Sami, R. and Twigg, A., "Lower bounds for distributed markov chain problems," *CoRR*, vol. abs/0810.5263, 2008.

[134] Sarlos, T., Benczur, A., Csalogany, K., Fogaras, D., and Racz, B., "To randomize or not to randomize: Space optimal summaries for hyperlink analysis," in *In the 15th International World Wide Web Conference, WWW*, pp. 297–306, 2006.

[135] Sinclair, A. and Jerrum, M., "Conductance and the mixing property of markov chains; the approximation of the permanent resolved," in *STOC*, pp. 235–244, 1988.

[136] Spielman, D. A. and Teng, S.-H., "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *STOC*, pp. 81–90, 2004.

[137] Thorup, M. and Zwick, U., "Approximate distance oracles," *J. ACM*, vol. 52, no. 1, pp. 1–24, 2005.

[138] Vitter, J. S., "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985. Also appeared in FOCS'83.

[139] Wicks, J. and Greenwald, A. R., "Parallelizing the computation of pagerank," in *Proc. 5th Workshop On Algorithms And Models For The Web-Graph (WAW)*, pp. 202–208, 2007.

[140] Wilson, D. B., "Generating random spanning trees more quickly than the cover time," in *STOC*, pp. 296–303, 1996.

[141] Woodruff, D. P., "Optimal space lower bounds for all frequency moments'," in *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pp. 167–175, 2004.

[142] Yao, A. C.-C., "Some complexity questions related to distributive computing (preliminary report)," in *STOC*, pp. 209–213, 1979.

[143] Zhong, M. and Shen, K., "Random walk based node sampling in self-organizing networks," *Operating Systems Review*, vol. 40, no. 3, pp. 49–55, 2006.

[144] Zhong, M., Shen, K., and Seiferas, J. I., "Non-uniform random membership management in peer-to-peer networks," in *INFOCOM*, pp. 1151–1161, 2005.